University of Sheffield

COM3529 Software Testing and Analysis

Larger Tests

Professor Phil McMinn

Larger Tests

Even with mocking, unit testing cannot serve all of our needs.

To check the behaviour of our code with that of others – or external services our code needs to work with, **integration tests** are necessary.

For test the software as a whole, however, we need **system tests**.

There are a host of other reasons why we might want to write larger tests.

Why Write Larger Tests?

1. Fidelity
2. Unfaithful Doubles
3. Configuration Issues
4. Issues that arise under load
5. Unanticipated behaviours, inputs and side effects
6. Emergent Behaviours

Fidelity – Realism of the Environment

Fidelity is the property by which a test is reflective of the real behaviour of the system under test.

One way of envisioning fidelity is in terms of the **environment**:

- **Unit tests bundle a test and a small portion of the code together as a runnable unit, which ensures the code is tested, but is very different from how the production code runs.**
- Production is the environment of highest fidelity in testing, but testing should not be carried out there! The best alternative is a **staged version of the system**, i.e., the system in its actual environment where testing can take place without using (or losing) real data.

Fidelity – Realism of Test Data

Tests can also be measured in terms of how faithful the test content is to reality.

Many handcrafted tests are dismissed by engineers if the test data itself looks unrealistic.

Test data copied from production is more faithful to reality, having being captured that way.

A big challenge is how to create realistic test traffic before launching new code!

Unfaithful Doubles

Doubles do not exactly replicate the thing they are doubling.

This can lead to gaps in unit tests.

Furthermore, doubles and their originals can become out of sync, particularly whether the author of the original class is not also in charge of the double.

Configuration Issues

Unit tests cannot test configuration issues related to the whole system.

At Google, configuration changes are the number one reason for many major outages the company has experienced.

> **In 2013 there was a global Google outage because a bad network configuration was pushed that was never tested.**
>
> `https://www.wired.co.uk/article/googledip`

Issues that Only Arise Under Load

Unit tests cannot check for system performance and particularly its performance under extreme loads (this is called **stress testing**).

Large volumes are big! Often thousands or millions of queries per second.

Unanticipated behaviours, inputs and side effects

Unit tests are limited by the imagination of the engineer who wrote them!

Issues that users find with a product are mostly unanticipated (else they would have been tested for!)

This is where **automated testing *techniques*** (and AI) become useful – for example, **fuzzing**.

Emergent Behaviours

Unit tests are limited to the scope that they cover, so if behaviour changes outside of this scope it cannot be detected.

… and because unit tests are designed to be fast and reliable, they deliberately eliminate the chaos of real dependencies, network and data.

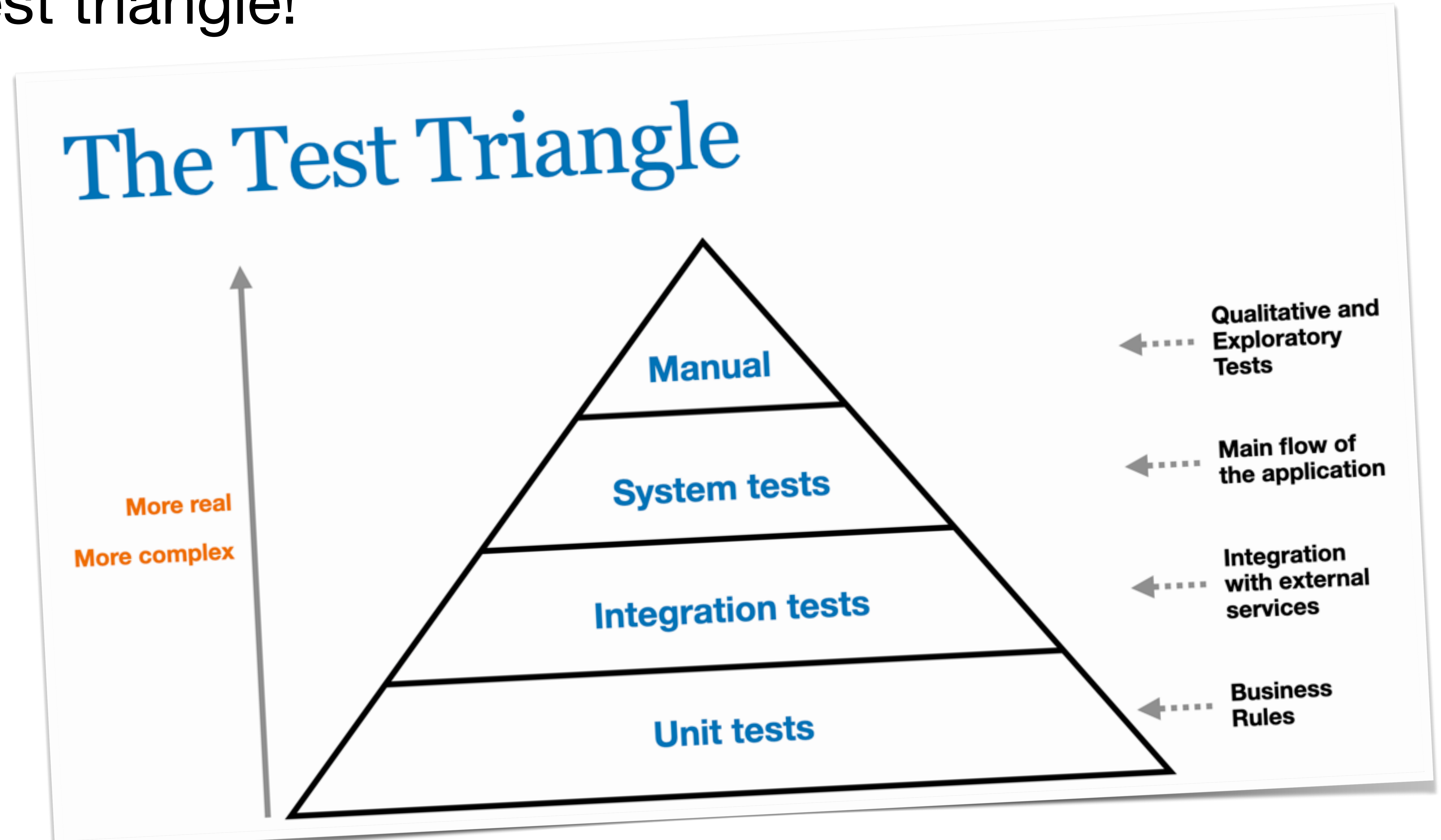
Unit tests are like problems in theoretical physics … ensconced in a vacuum, neatly hidden from the mess of the real world.

This is great for speed and reliability – **but misses certain types of defects.**

These reasons are *not* to say “don’t do unit testing”

But rather that **larger tests should be used to complement unit tests.**

Recall the test triangle!



Some Types of System Test

Browser and Device Testing

Browsers and devices vary, and often dictate how users interact with the application.

For example, what/how much content can be rendered, given the screen size.

Getting hold of all of these versions of devices can be problematic – leading to the setting up of public device libraries.

Performance, Load and Stress Testing

These test are critical for **ensuring that there is no degradation in performance between versions and that the system can handle expected spikes in traffic**.

Deployment Configuration Testing

Often it is not the code that is the source of defects, but the configuration:

- Data files
- Databases
- Option definitions

Larger tests can test these things because the configuration files are read during the launch of the product's binary.

Often these tests are smoke tests of the system, without needing much by way of additional test data or verification via assertions: **If the system starts successfully, the test passes!**

Exploratory Testing

Exploratory testing is a form of manual testing that focuses on looking for questionable behaviours by trying out new and established user scenarios.

Trained users/testers interact with a a product, looking for new paths through the system for which behaviour deviates from expected or intuitive behaviour – or if there are security vulnerabilities.

User Acceptance Testing

Unit tests are written by developers, making it quite likely that misunderstandings about the intended behaviour of the system are reflected not only in the code but also the unit tests.

User Acceptance Tests exercise the system to ensure the overall behaviour for specific user journeys is as intended.

Such tests are said to run the system from “end to end” and are sometimes called end-to-end (E2E) tests.

Multiple frameworks exist, such as Cucumber and RSpec, to make tests writeable and readable in a user-friendly language.

Disaster Recovery

These tests expose how well the system reacts to unexpected changes or failures such as datacentre fires, malicious attacks, earthquakes etc.

Systems are taken out or taken down to observe the effects of what happens in practice.

Often this tests the fragility of the organisation too – as key decision makers may suddenly be out of contact.

Chaos Engineering

Made popular by Netflix, chaos engineering tests the resilience of systems by writing programs that simulate turbulent conditions in practice.

These programs introduce faults/problems that include server shutdowns, latency injections, and resource exhaustion.

These faults are higher level and more targeted than those introduced by *mutation analysis*, which we will cover later in the course.

Problems with Larger Tests

Brittleness in Larger Tests

Larger tests do not escape from the problem of brittleness.

Recall that brittleness is when tests fail due to internal changes in code (refactoring) that do not affect its external behaviour.

Example:

> In end-to-end system testing, the tests could be too closely couple to the names of GUI elements or their layout – meaning that test fails then the UI changes.

Flaky Tests

Larger tests are more prone to suffer from the problem of flakiness.

Flakiness is a result of non-determinism in tests or code that cause tests to fail, when actually they work (and less common: vice versa).

Example:

> A system test that executes some code that requires a response from a server. When the server is running, the test passes, but when the server is down, the test fails. However, when the test fails, there's no fault in the code.

Unit tests can also be flaky, but since they tend to focussed on logic and not involve non-deterministic elements, this is less common.

Problems Caused by Flaky Tests

The intermittency of flaky test failures is maddening for developers because the source of flakiness is **often hard to track down**:

> The symptoms of the problem (a failing test) are often far removed from the source of the non-deterministic behaviour, particularly with system tests.

But as soon as a test suite has several flaky tests, **developers tend to stop trusting it and/or running it**.

Flaky tests are a **frequent factor** in failing builds on continuous integration servers.

Mitigations for Flaky Tests

Re-run until pass

Lengthen timeouts

Mock external dependencies

Fix the underlying issue!

Why might I be a *Flaky Test*?

I test some some code that relies on what day of the week it is.

Why might I be flaky?

I test some some code that involves a number of threads being started and executed.

Why might I be flaky?

I test some some code that is dependent on some tasks being executed in a specific timeframe.

Why might I be flaky?

I test some some code that checks for the latest news items.

Why might I be flaky?

Problems with Larger Tests

1. **Reliability** – larger tests can be *brittle* and *flaky*
2. **Speed** – larger tests can be slow, interrupting a developer's workflow
3. **Ownership** – unit tests have an owner, who is in charge of system tests?
4. **Standardisation.** Unit tests have a standard frameworks and form. Larger tests have to cope with different infrastructures and environments, with multiple competing tools and frameworks (some of which are developed in house).

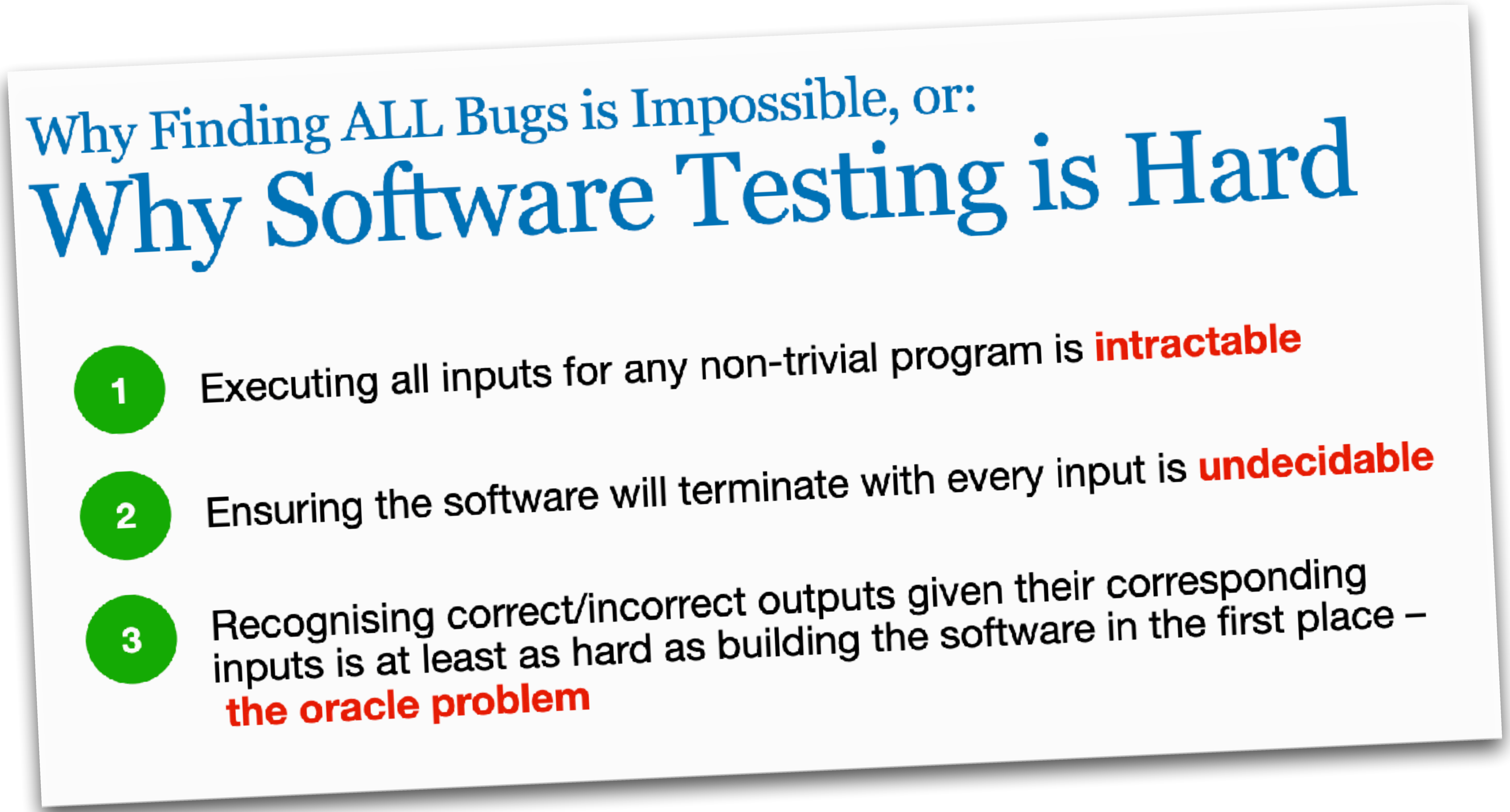
University of Sheffield

COM3529 Software Testing and Analysis

An Introduction to Coverage Criteria

Professor Phil McMinn

Recap from Week 1



Why Finding ALL Bugs is Impossible, or:

Why Software Testing is Hard

1. Executing all inputs for any non-trivial program is **intractable**
2. Ensuring the software will terminate with every input is **undecidable**
3. Recognising correct/incorrect outputs given their corresponding inputs is at least as hard as building the software in the first place – **the oracle problem**

A Testing Problem

To: p.mcminn@sheffield.ac.uk
From: student3529@sheffield.ac.uk
Subject: A Problem with Testing – Please help!!!

Dear Phil

You’ve told us how to write tests, but in the first lecture you told us that we cannot try all inputs or try to test everything.

But I’m really stuck – how to I decide what to actually test?

Yours,
Stu

A Testing Solution

To: student3529@sheffield.ac.uk
From: p.mcminn@sheffield.ac.uk
Subject: Re: A Problem with Testing – Please help!!!

Dear Stu,

Have no fear.

In the next lecture I'm going to introduce **coverage criteria**, which help you decide what to test, and what you've missed.

Be sure to be there!

Best,
Phil

Coverage Criteria

A **coverage criterion** takes an abstract representation of a piece of software and **divides it up** into **testable features**.

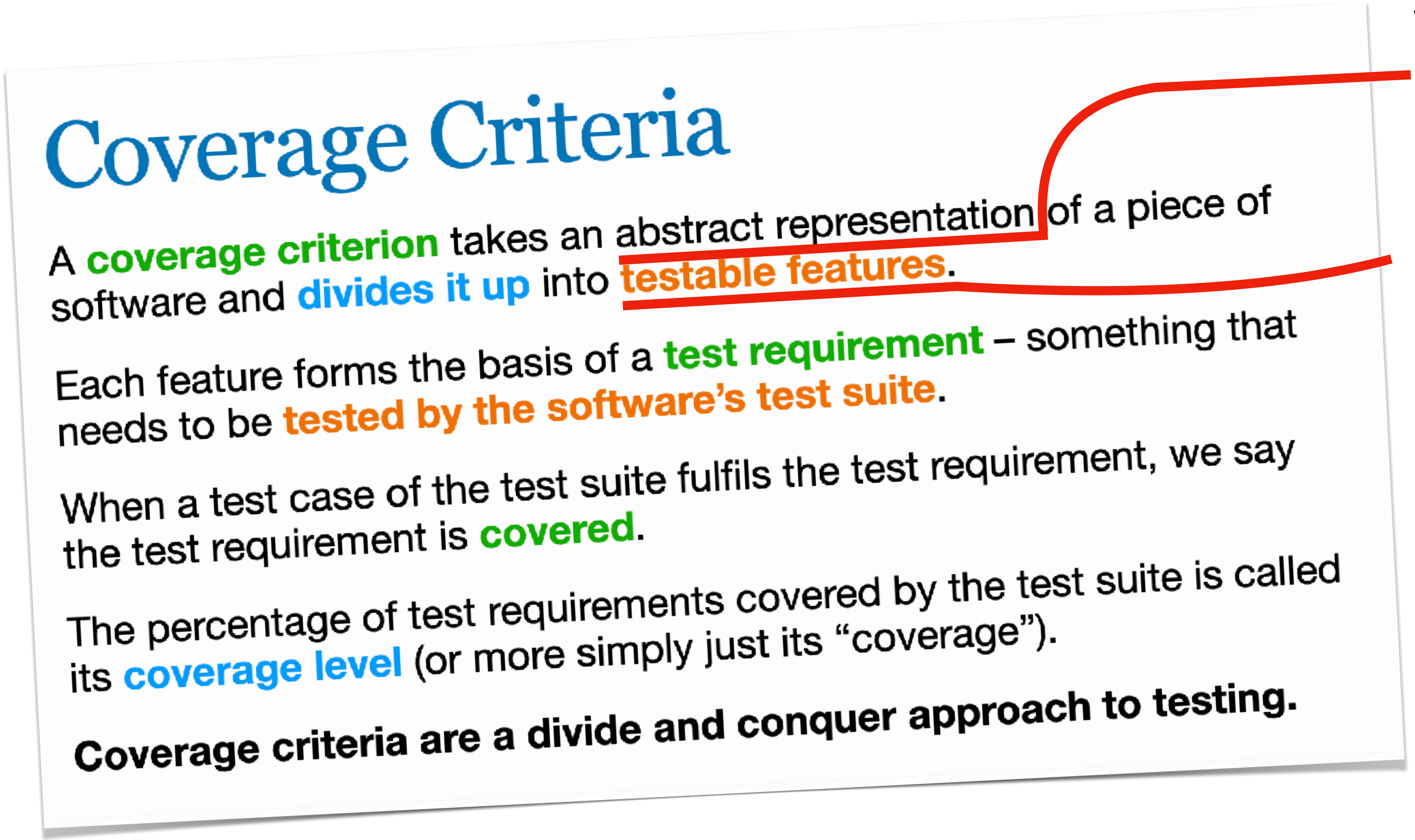
Each feature forms the basis of a **test requirement** – something that needs to be **tested by the software's test suite**.

When a test case of the test suite fulfils the test requirement, we say the test requirement is **covered**.

The percentage of test requirements covered by the test suite is called its **coverage level** (or more simply just its "coverage").

Coverage criteria are a divide and conquer approach to testing.

Some Obvious Questions



Coverage Criteria

A **coverage criterion** takes an abstract representation of a piece of software and **divides it up** into **testable features**.

Each feature forms the basis of a **test requirement** – something that needs to be **tested by the software's test suite**.

When a test case of the test suite fulfils the test requirement, we say the test requirement is **covered**.

The percentage of test requirements covered by the test suite is called its **coverage level** (or more simply just its "coverage").

Coverage criteria are a divide and conquer approach to testing.

What do you mean by "abstract representation"?

What do you mean by "testable features"?

Statement Coverage

One **very simple representation** is the **program statements** that make up a piece of software.

Each **feature** is a program statement.

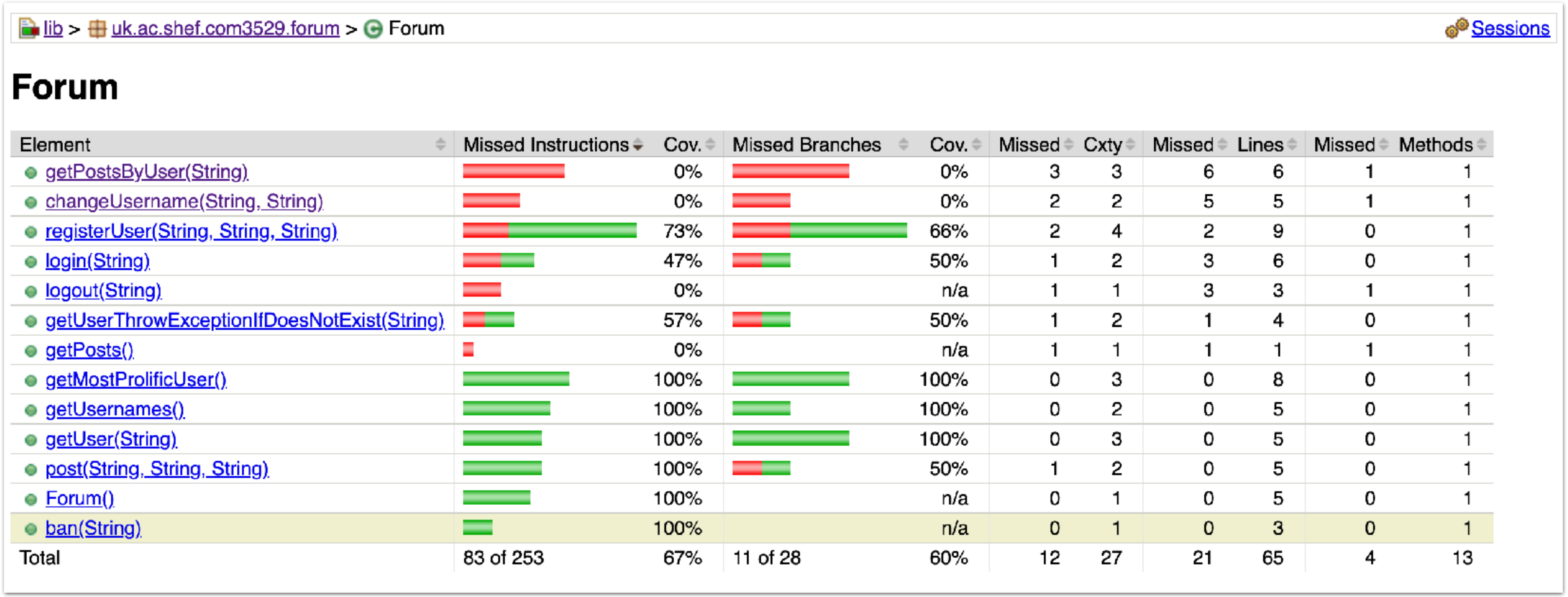
Each test requirement involves executing (“covering”) each line.

Statement coverage is the percentage of lines of code executed by the test suite.

JaCoCo

Run `./gradlew jacocoTestReport` at the command line after executing your tests.

Go into the `code/lib/build/reports/jacoco` directory

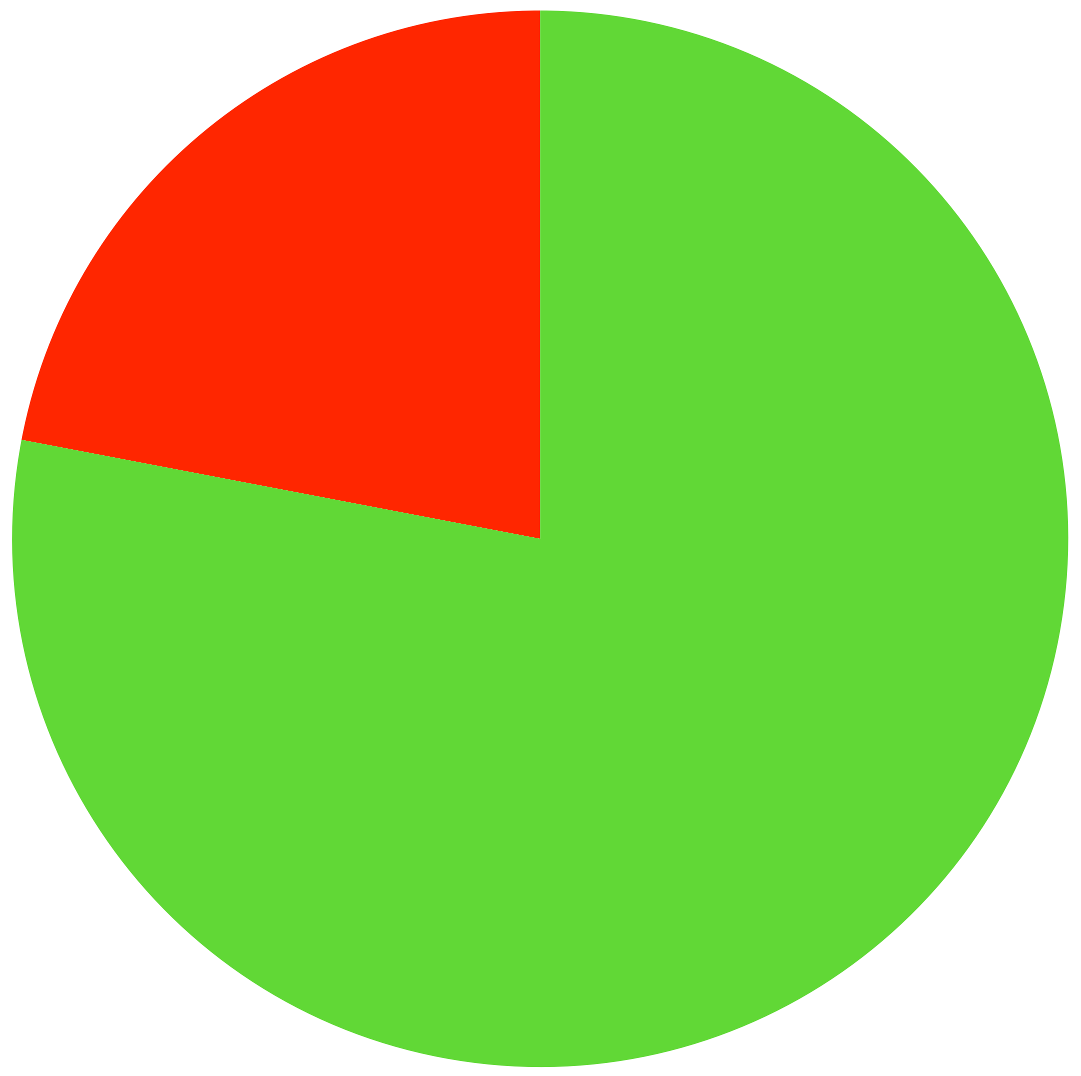


lib > uk.ac.shef.com3529.forum > Forum Sessions

Forum

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getPostsByUser(String)		0%		0%	3	3	6	6	1	1
changeUsername(String, String)		0%		0%	2	2	5	5	1	1
registerUser(String, String, String)		73%		66%	2	4	2	9	0	1
login(String)		47%		50%	1	2	3	6	0	1
logout(String)		0%		n/a	1	1	3	3	1	1
getUserThrowExceptionIfDoesNotExist(String)		57%		50%	1	2	1	4	0	1
getPosts()		0%		n/a	1	1	1	1	1	1
getMostProlificUser()		100%		100%	0	3	0	8	0	1
getUsernames()		100%		100%	0	2	0	5	0	1
getUser(String)		100%		100%	0	3	0	5	0	1
post(String, String, String)		100%		50%	1	2	0	5	0	1
Forum()		100%		n/a	0	1	0	5	0	1
ban(String)		100%		n/a	0	1	0	3	0	1
Total	83 of 253	67%	11 of 28	60%	12	27	21	65	4	13

Coverage Level



JaCoCo measures **instructions** as opposed to statements because it computes coverage at the level of Java ByteCode

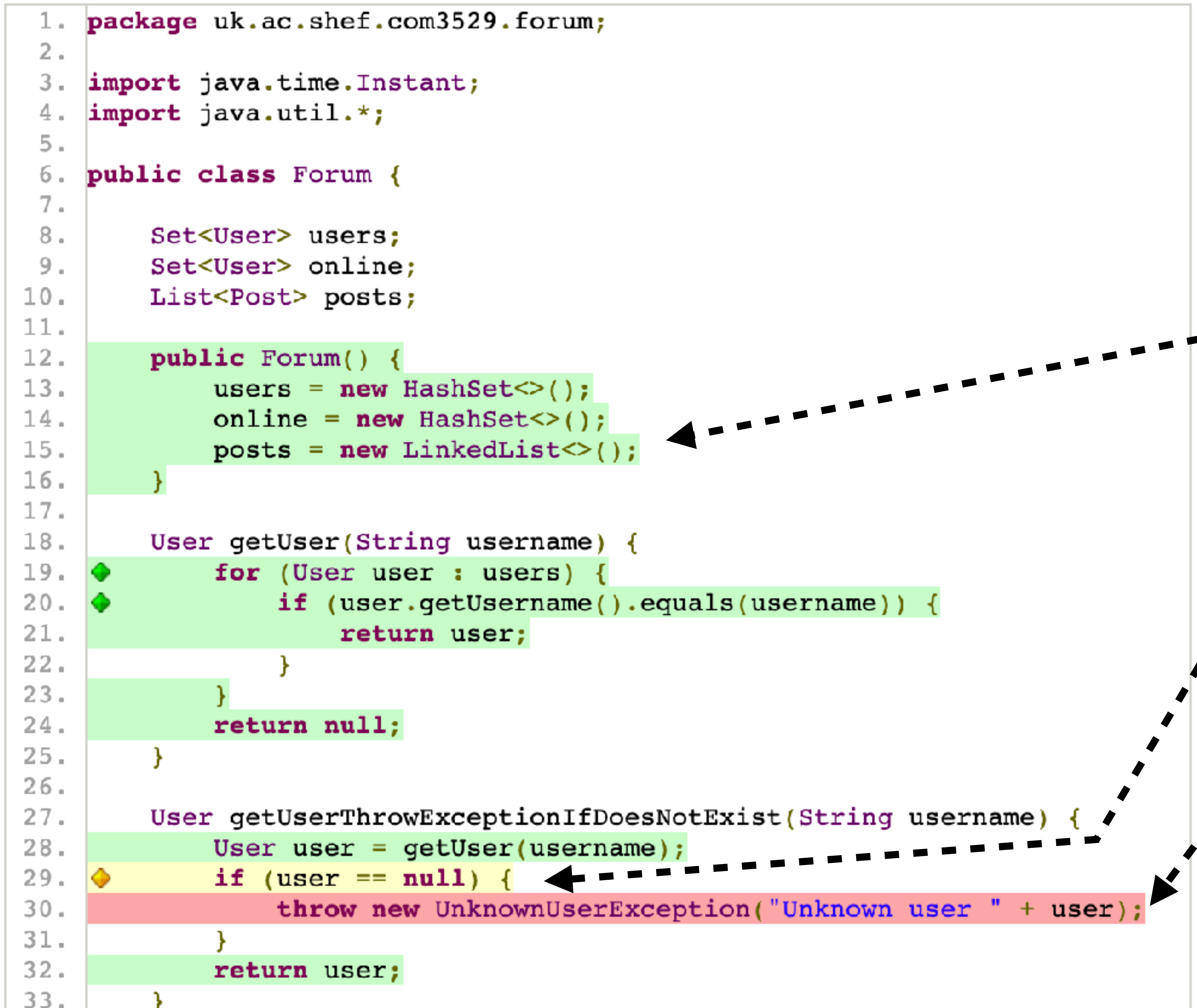
78% of statements executed (covered)

22% of statements not executed (uncovered)

The coverage level (or just "coverage") is the percentage of test requirements fulfilled by the test suite.

Here, 78% statements are executed, so the **statement coverage is 78%**.

JaCoCo Class Coverage Report



```
package uk.ac.shef.com3529.forum;

import java.time.Instant;
import java.util.*;

public class Forum {

    Set<User> users;
    Set<User> online;
    List<Post> posts;

    public Forum() {
        users = new HashSet<>();
        online = new HashSet<>();
        posts = new LinkedList<>();
    }

    User getUser(String username) {
        for (User user : users) {
            if (user.getUsername().equals(username)) {
                return user;
            }
        }
        return null;
    }

    User getUserThrowExceptionIfDoesNotExist(String username) {
        User user = getUser(username);
        if (user == null) {
            throw new UnknownUserException("Unknown user " + user);
        }
        return user;
    }
```

Green = **executed** by test(s)

Yellow = one ***branch*** (true or false) of an `if` or `for`/`while` statement executed by test(s)

Red = **not executed** by test(s)

Criticisms and Other Coverage Criteria

Statements are one very simple way of representing the software and dividing it up (and then “conquering”) for testing.

Statement Coverage is not that useful a method of “deciding what to test”, more a way of discovering what is *not* covered by a test suite, and deciding whether more test cases are needed.

We will meet different (and potentially more useful) coverage criteria throughout the module.

Does Obtaining 100% Coverage Ensure our Program is Bug-Free?

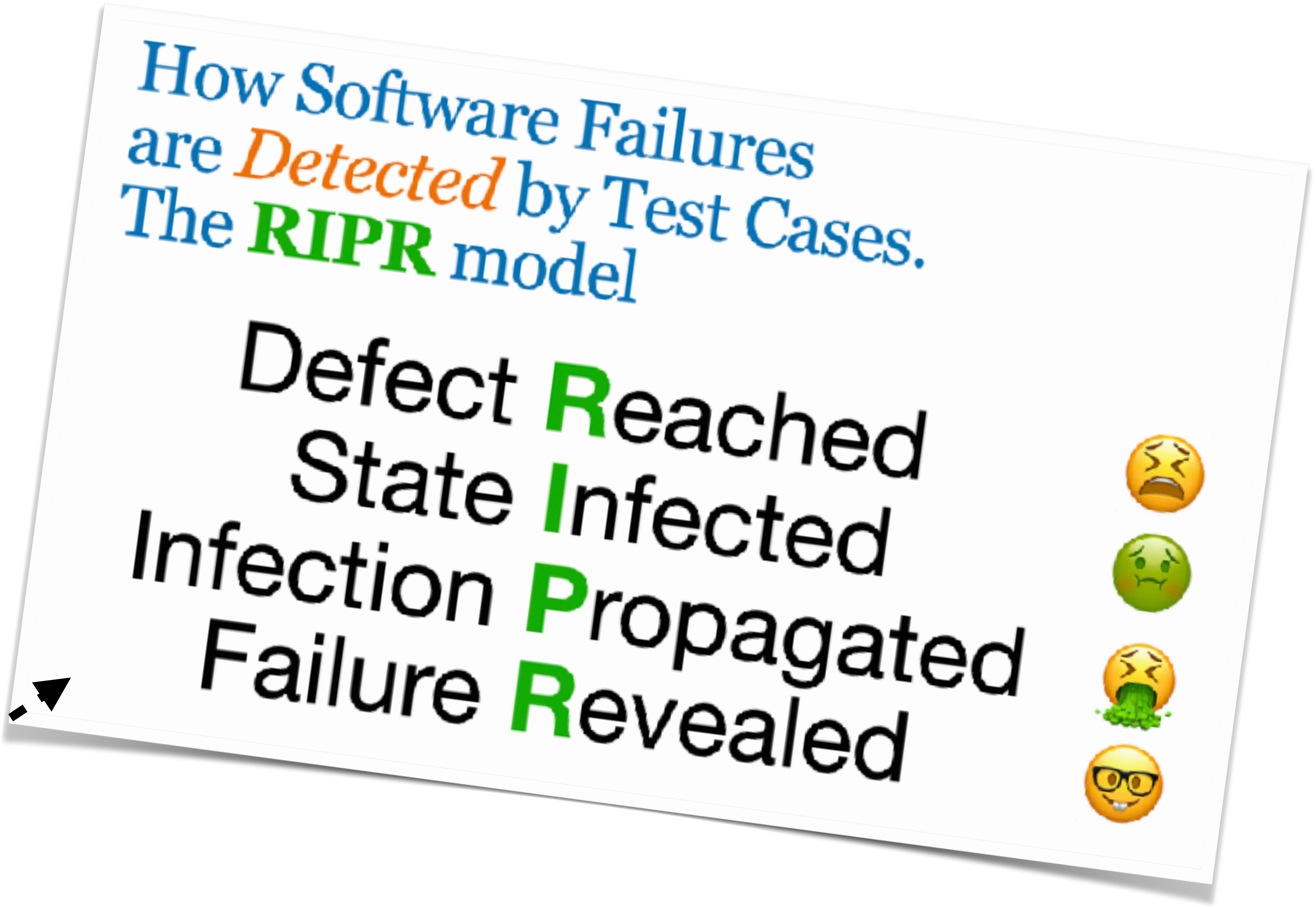
Does Obtaining 100% Coverage Ensure our Program is Bug-Free?

NO! – coverage criteria, such as statement coverage, are just a method of generating test requirements in a systematic way.

For example, 100% statement coverage might execute the *majority* of defects, but:

- **does not** guarantee that the defects will infect the program's state
- **does not** guarantee infections will propagate to the program's output as failures
- **does not** guarantee failures will be caught by the test suite's assertions

Recall the RIPR model from week 1



How Software Failures are Detected by Test Cases.
The RIPR model

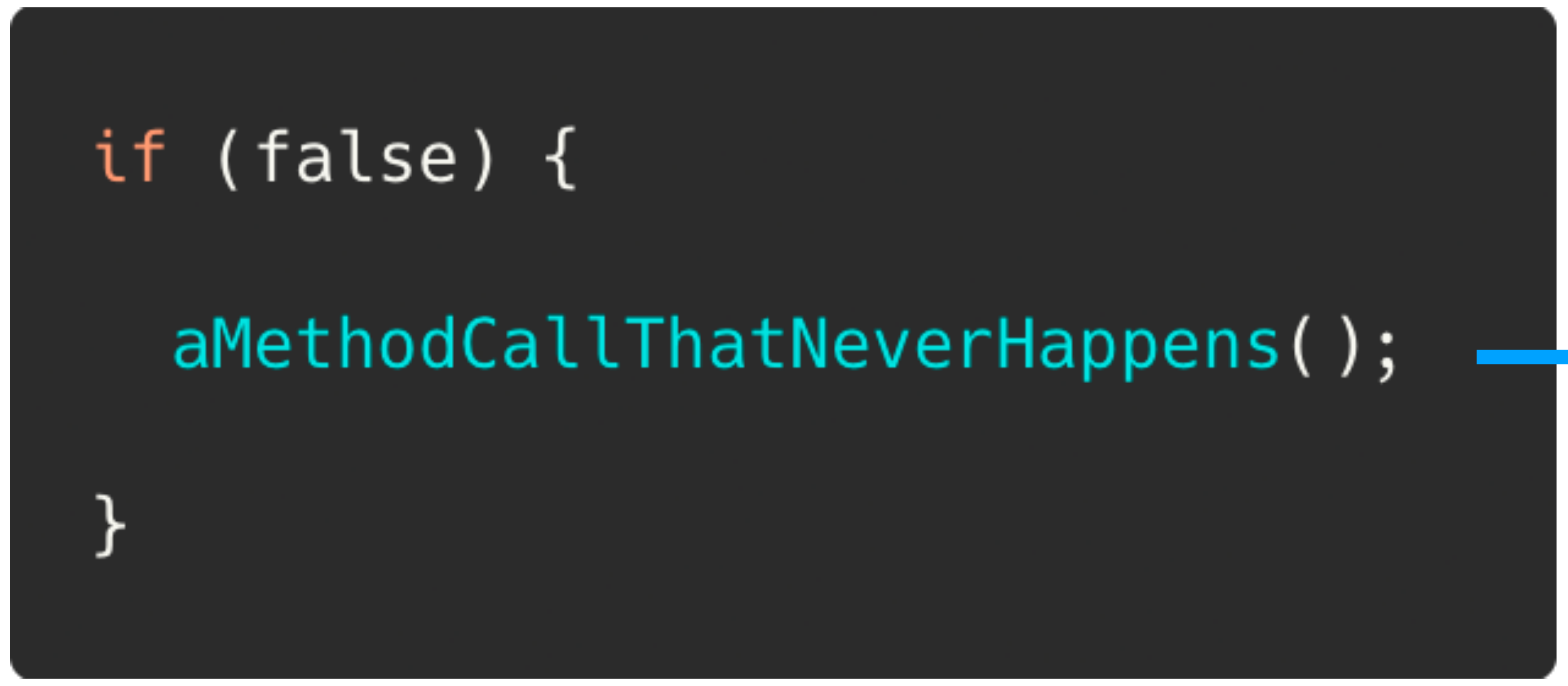
Defect Reached
State Infected
Infection Propagated
Failure Revealed

Infeasible Test Requirements

Moreover some test requirements may be **infeasible.**

This means they are impossible to cover.

For example, a line of code that is impossible to execute – i.e., dead code.



```
if (false) {
  aMethodCallThatNeverHappens();
}
```

This program statement can never be executed! It forms an infeasible coverage requirement for statement coverage.

Recap: Important Terminology

Coverage Criterion: A method of dividing up software into a set of test requirements for testing.

Test Requirement: A feature of a piece of software that the test suite is obliged to fulfil by a coverage criterion. Note a test requirement is not the same as software requirement and is not the same as a test case. A test case could fulfil several test requirements.

Infeasible Test Requirement: A test requirement that is impossible to fulfil.

Coverage Level: The percentage of test requirements executed (covered) by a test suite.

White, Black, and Grey Box Coverage Criteria

One way of categorising coverage criteria is to distinguish whether the test requirements are derived from the code or whether it makes no reference to the way that it has been programmed.

White Box

Full knowledge of internal code structure.
Are often called "code coverage" criteria for this reason.

Black Box

No knowledge of internal code structure.
Coverage criteria based on requirements, designs, or abstract models of the software.

Grey Box

Some knowledge of internal code structure.
Coverage criteria based on a mix of artefacts considered "white box" and "black box".

Different Criteria, Different Defects

Defect of **commission**

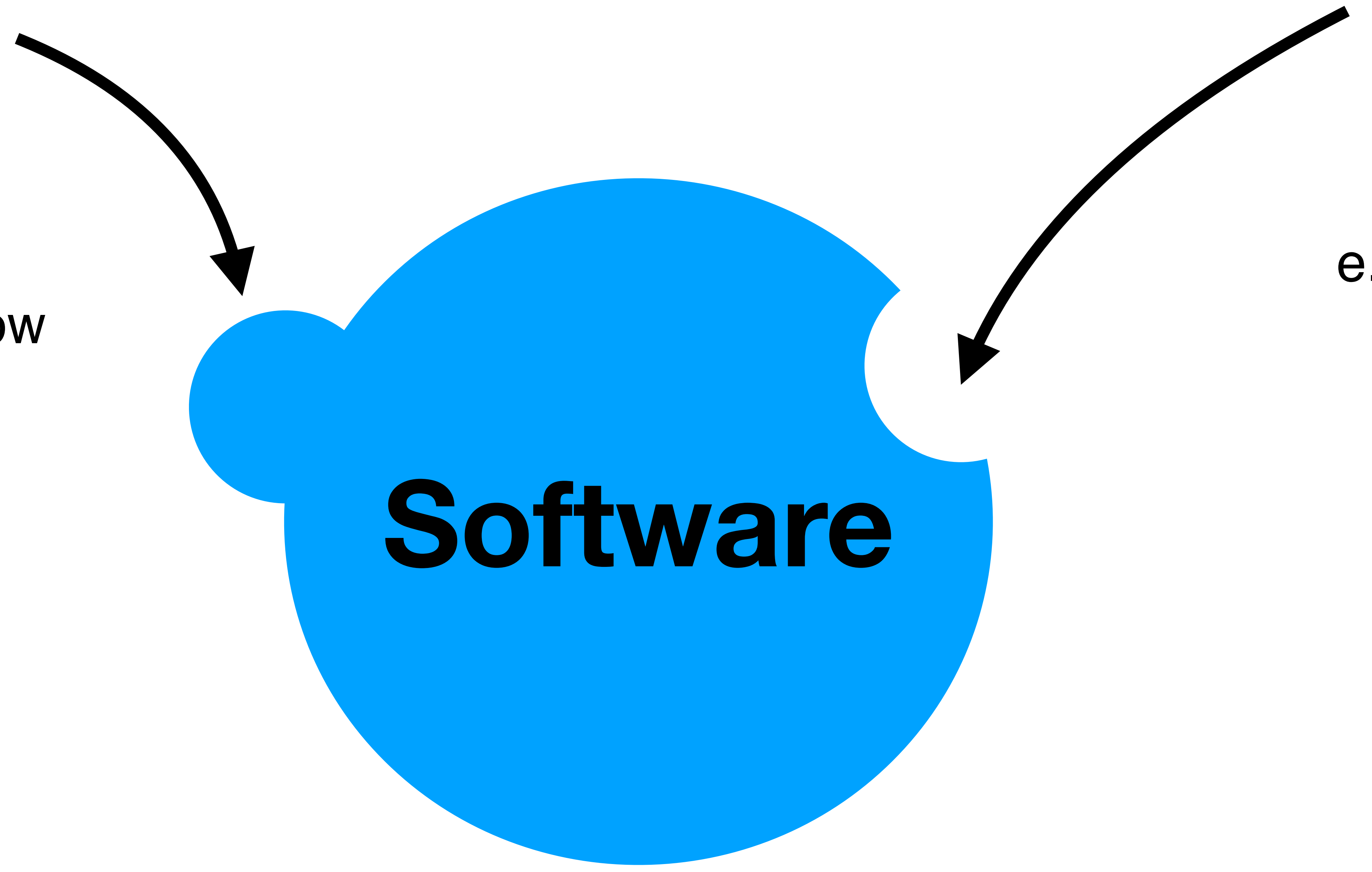
Code does something *more* than it is supposed to

e.g., a C program iterating beyond the end of an array, causing a buffer overflow

Defect of **omission**

Code does something *less* than it is supposed to

e.g., an unimplemented requirement



White Box Coverage Criteria

Best suited to detecting defects of commission as can "see" into code.

Less well suited to detecting defects of omission as no knowledge of software requirements.

Black Box Coverage Criteria

Less well suited to detecting defects of commission as no knowledge of internal workings of the software.

Best suited to detecting defects of omission as based on requirements, abstract models of software.