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.

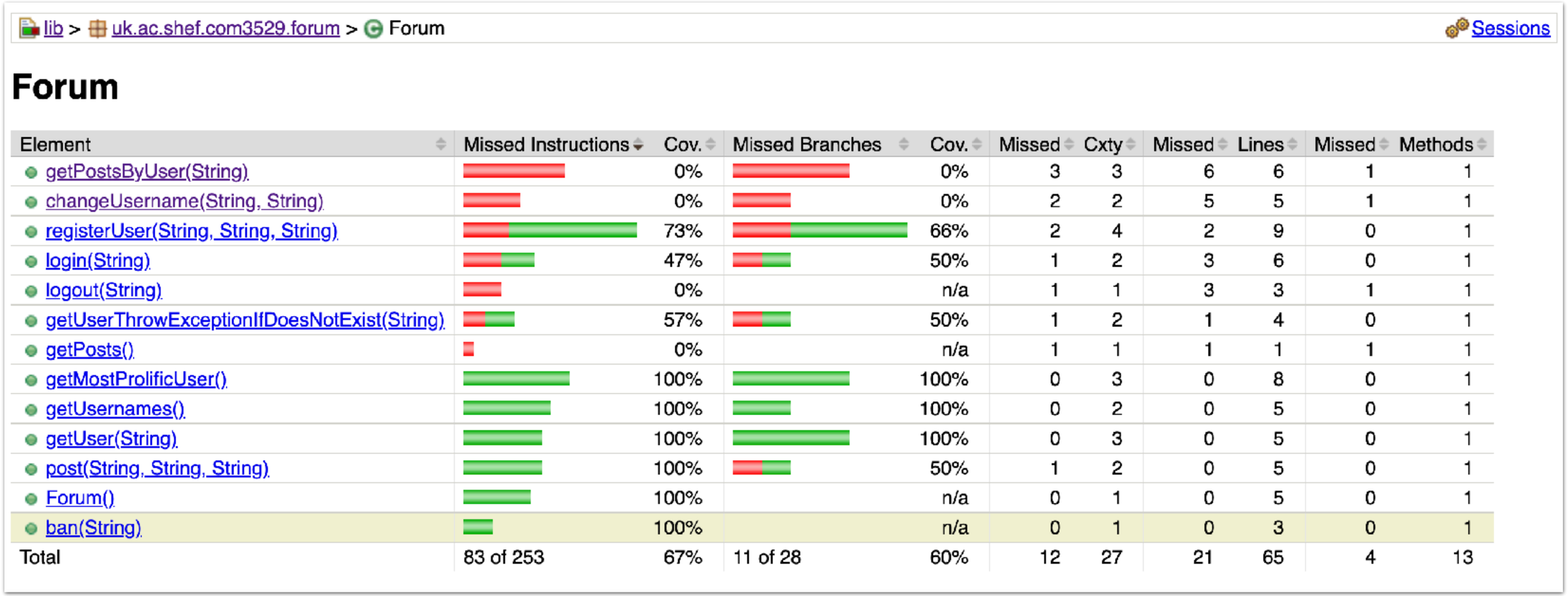
Each test requirement involves executing (“covering”) each line.

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



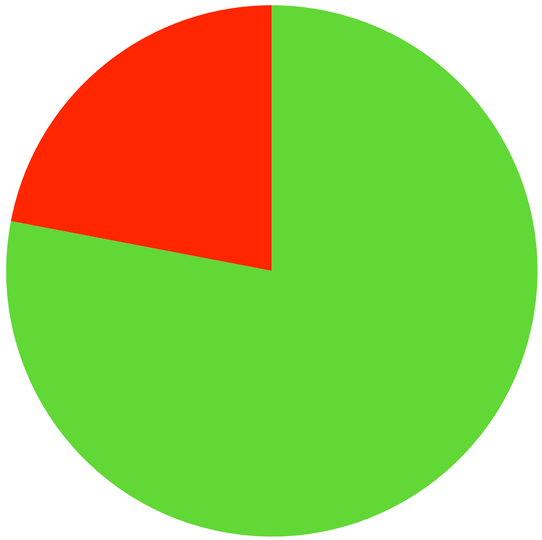
lib > uk.ac.shef.com3529.forum > Forum

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

```
1.  package uk.ac.shef.com3529.forum;
2.
3.  import java.time.Instant;
4.  import java.util.*;
5.
6.  public class Forum {
7.
8.      Set<User> users;
9.      Set<User> online;
10.     List<Post> posts;
11.
12.     public Forum() {
13.         users = new HashSet<>();
14.         online = new HashSet<>();
15.         posts = new LinkedList<>();
16.     }
17.
18.     User getUser(String username) {
19.         for (User user : users) {
20.             if (user.getUsername().equals(username)) {
21.                 return user;
22.             }
23.         }
24.         return null;
25.     }
26.
27.     User getUserThrowExceptionIfDoesNotExist(String username) {
28.         User user = getUser(username);
29.         if (user == null) {
30.             throw new UnknownUserException("Unknown user " + user);
31.         }
32.         return user;
33.     }
```

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.

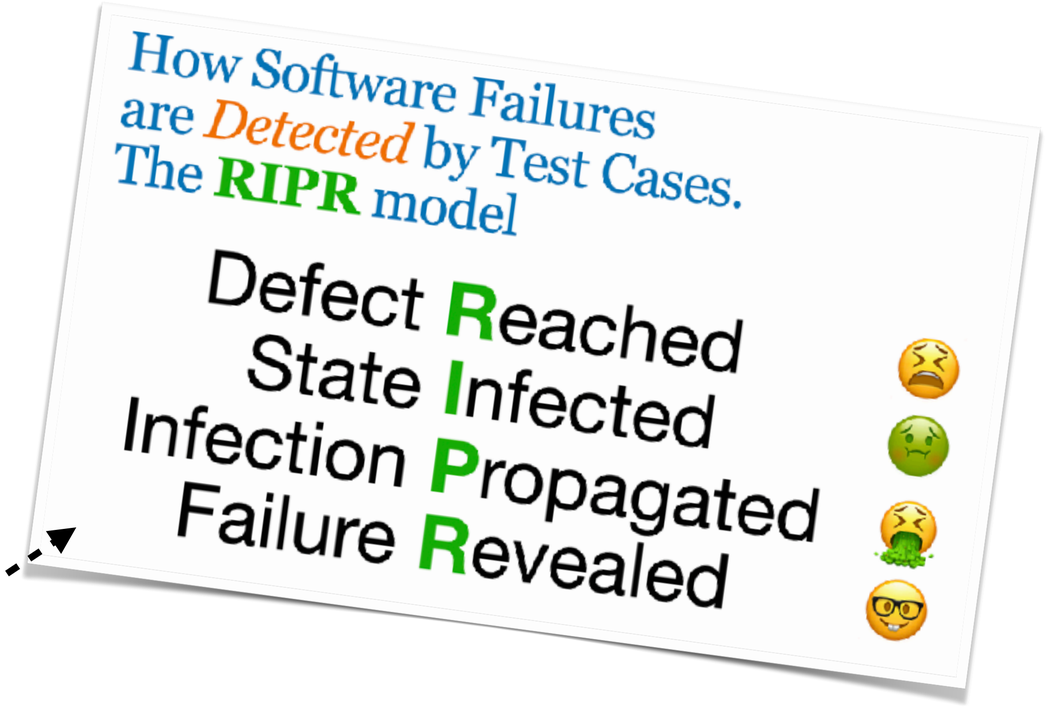
Does Obtaining 100% Coverage Ensure our Program is Bug-Free?

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



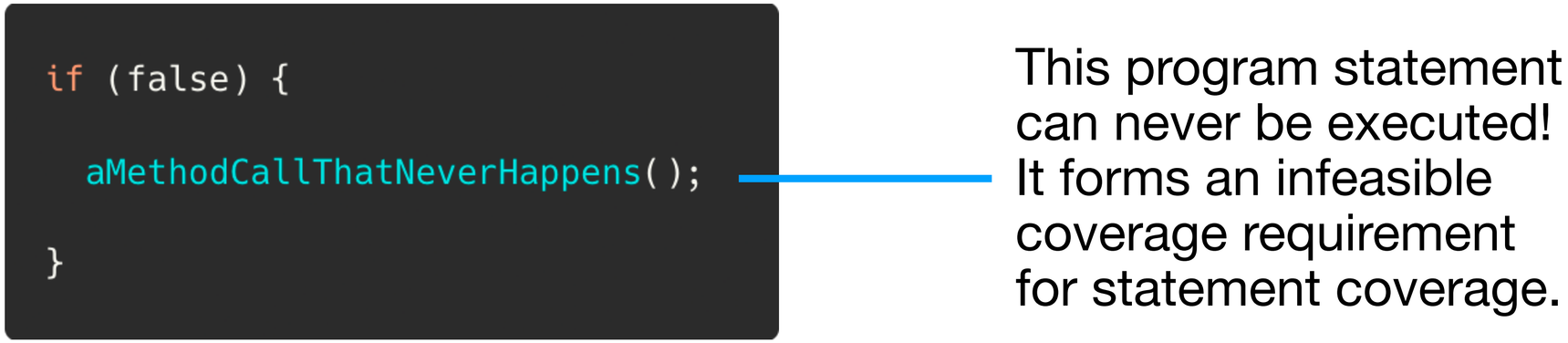
Recall the RIPR model from week 1

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.



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**

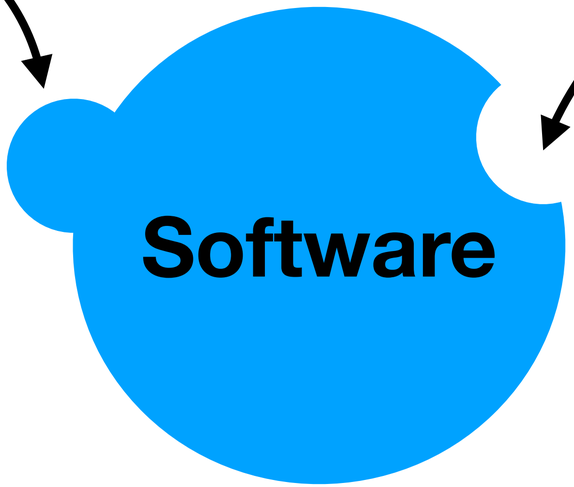
Code does something *more* than it is supposed to

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.