University of Sheffield

An Introduction to Software Testing

https://github.com/philmcminn/eyup-testing

Phil McMinn

About me

1990s

2000s

2010s

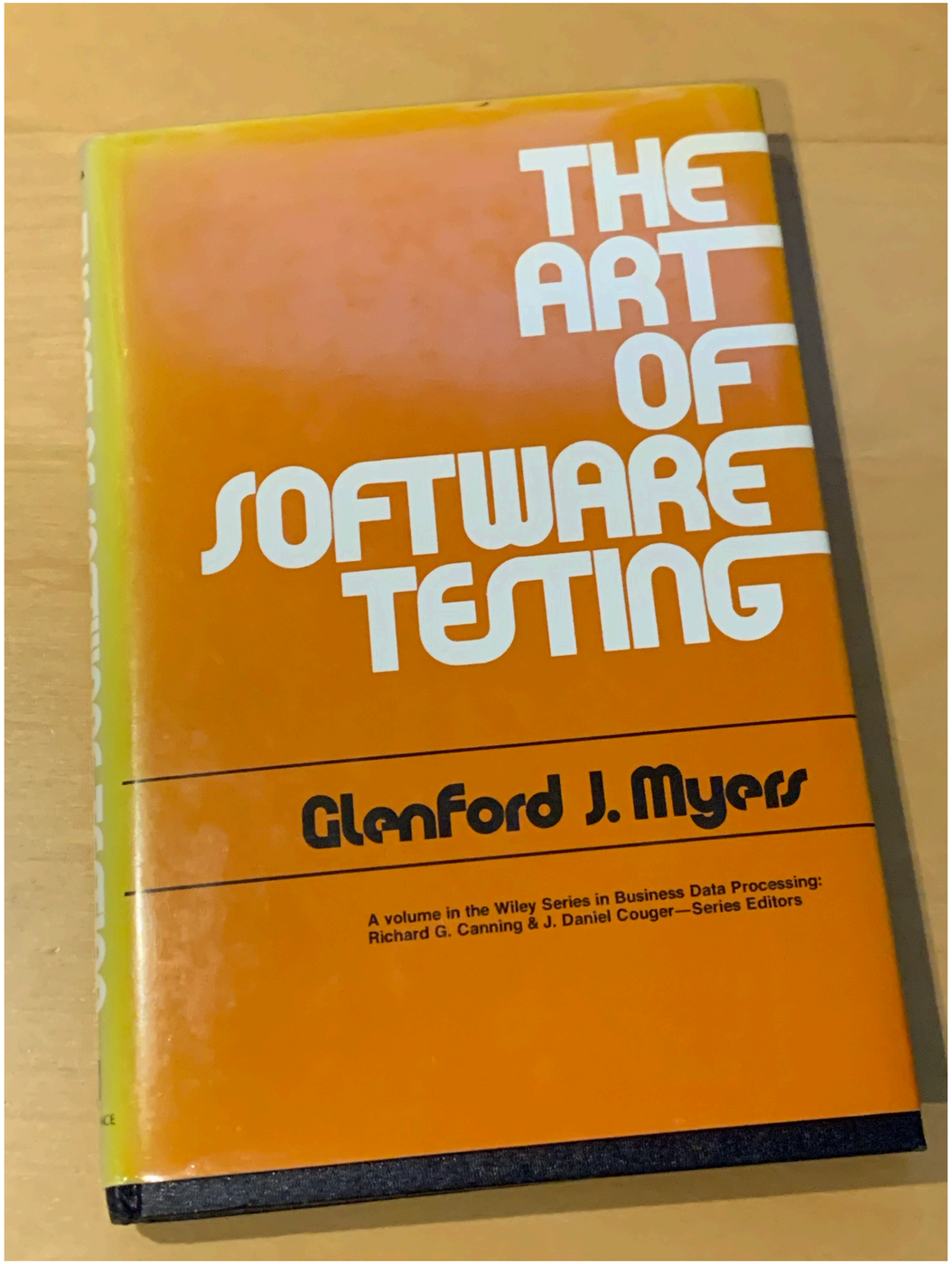
The Triangle Problem

You need to test a program:

The program reads three integer values from the console. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles or equilateral.

Write down, on a sheet of paper, the test cases you feel would adequately test this program.

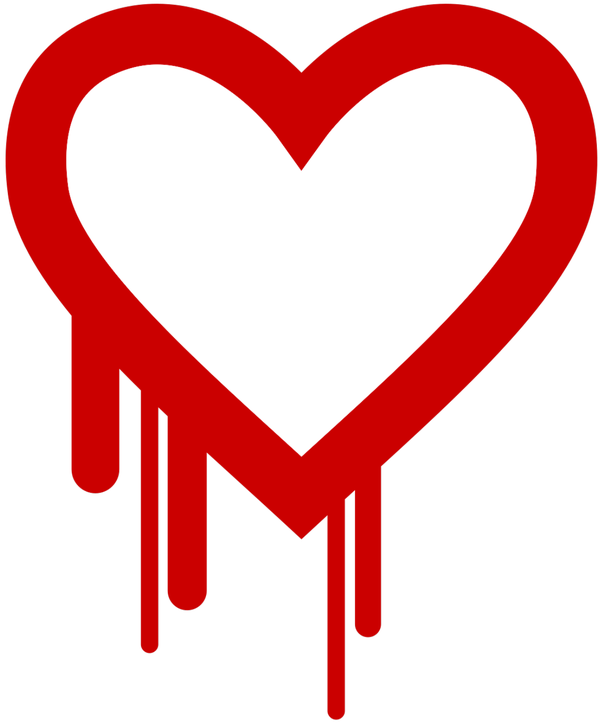
(Note that an equilateral triangle has three equal sides; an isosceles, two; and a scalene, none.)

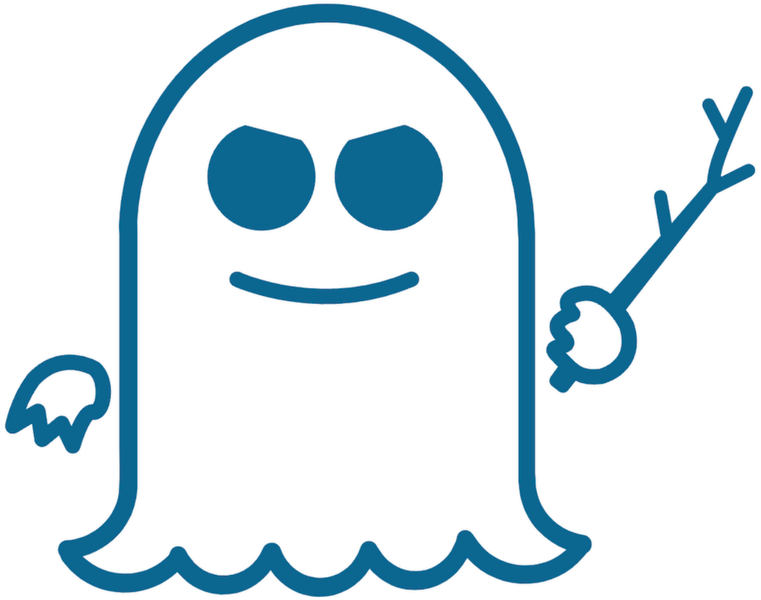


1. Do you have a test case that represents a *valid* scalene triangle? (Note that test cases such as 1,2,3 and 2,5,10 do not warrant a "yes" answer, because there does not exist a triangle having such sides.)
2. Do you have a test case that represents a valid equilateral triangle?
3. Do you have a test case that represents a valid isosceles triangle? (A test case specifying 2,2,4 would not be counted.)
4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (e.g., 3,3,4; 3,4,3; and 4,3,3)?
5. Do you have a test case in which one side has a zero value?
6. Do you have a test case in which one side has a negative value?
7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third? (That is, if the program said that 1,2,3 represents a scalene triangle, it would contain a bug.)
8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (e.g., 1,2,3; 1,3,2; and 3,1,2)?
9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (e.g., 1,2,4 or 12,15,30)?
10. Do you have at least three test cases in category 9 such that you have tried all three permutations (e.g., 1,2,4; 1,4,2; and 4,1,2)?
11. Do you have a test case in which all sides are 0 (i.e., 0,0,0)?
12. Do you have at least one test case specifying noninteger values?
13. Do you have at least one test case specifying the wrong number of values (e.g., two, rather than three, integers)?
14. For each test case, did you specify the expected output from the program in addition to the input values?


Uber

The **Post Office software scandal** was the subject of an episode of the **BBC's "Panorama"** and is available here:
https://www.bbc.co.uk/iplayer/episode/m0016t20/panorama-the-post-office-scandal





Who here thinks software testing is *important*?

Who here *likes* software testing?

Who here thinks testing is *hard*?

Who thinks testing and debugging are basically the same thing?

Who thinks that testing is to show that software *works*?

Who thinks that testing is to show that software *doesn't work*?

Who thinks that the idea behind testing is to *reduce risks* involved in using software?

Who thinks that the testing is to help in the development of *higher quality* software?

Beizer's Maturity Model

0. There's no difference between testing and debugging

1. The purpose of software testing is to show that software works

2. The purpose of software testing is show that software doesn't work

“Excellent testing can make you unpopular with almost everyone!”

— Bill McKeeman

Beizer's Maturity Model

0. There's no difference between testing and debugging

1. The purpose of software testing is to show that software works

2. The purpose of software testing is show that software doesn't work

3. The purpose of software testing is not to show anything in particular, but to reduce the risk of using software

4. Testing is a mental discipline that helps all IT professionals develop higher quality software

Why Software Testing is Hard

Exhaustive Testing

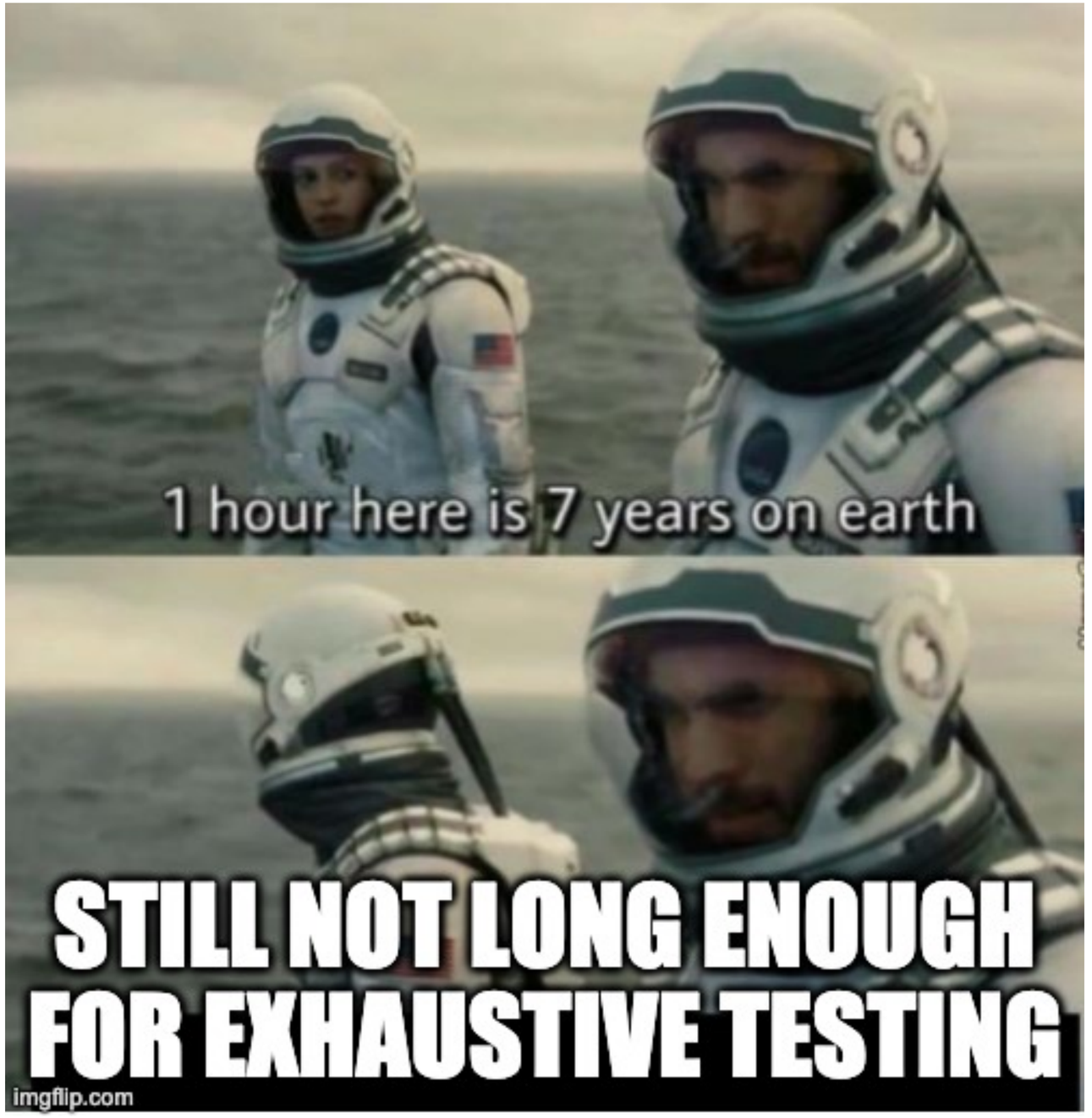
```
public class Calendar {

    public static int daysBetweenTwoDates(int year1, int month1, int day1,
                                          int year2, int month2, int day2) {
      // ...
```

The range of an int in Java is $-2{,}147{,}483{,}648$ to $2{,}147{,}483{,}647$ – or 2^{32}

With six ints that's 2^{32^6} or 2^{192} unique sets of parameters to try, which is approximately 6×10^{57} !

Suppose each possible input takes on average one nanosecond to execute. It would take 10^{41} years to try them all!


1 hour here is 7 years on earth
STILL NOT LONG ENOUGH
FOR EXHAUSTIVE TESTING
imgflip.com

The Halting Problem

The Halting Problem in Computer Science is basically the problem of not knowing if a program will terminate given some input.

If we give an arbitrary program an input, it has been proven that no program can be written that can say whether that original program will terminate.

And this is true in software testing: we don’t know if, given our test inputs, whether the program being tested will get stuck in an infinite loop!

Meaning that in general, exhaustive testing is not be just intractable it’s impossible too.

> Software testing can only show the presence, not the absence of bugs

— Edsger Dijkstra

But we don’t need every single one of those inputs to ensure the program is working… right …?

But **how do we choose** that subset of inputs?

This is the essence of the software testing problem.

We need to choose a set of inputs that will reveal as much information about the quality of the software as possible.

But **we don’t know** that we’ve selected all the inputs that will reveal all of the bugs.

The Oracle Problem

Even if we could

1) execute all software with all inputs

(i.e., if software testing was a **tractable** problem)

2) guarantee the software terminated with each input

(i.e., if software testing was a **computable** problem)

We would still need to solve the **oracle problem** – **how to know, given some input to a software system, that the output it gives is the correct one**

The Oracle Problem

In software testing an

oracle

is **something** or **someone** who can determine if a **software output is correct**

e.g. an **assertion in JUnit**, or
a **formal specification**

this is how **manual testing** is performed – a human being makes the judgment

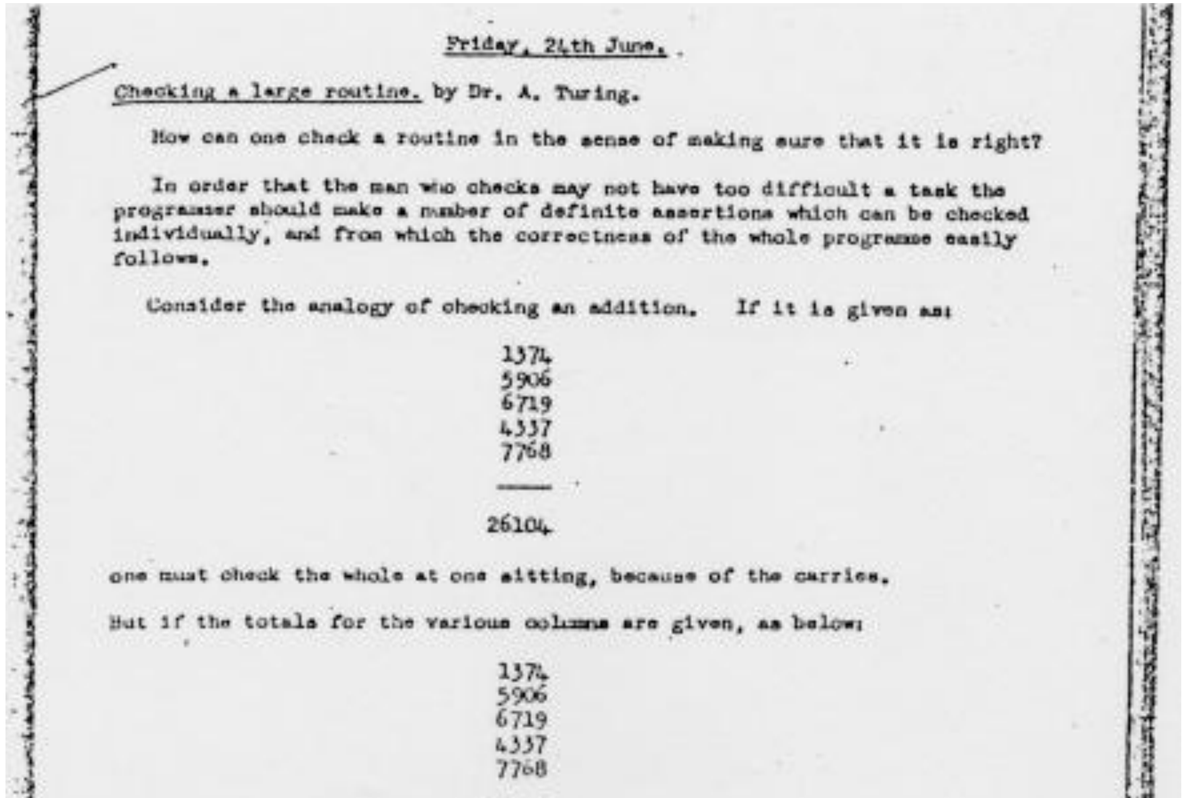
	Tractable problems	Intractable problems	Uncomputable problems
Description	Can be solved efficiently	Method for solving exists, but is hopelessly time consuming	Cannot be solved by any computer program
Computable in theory			
Computable in practice			
Example	Find the shortest route on a map	Decryption	**Finding all bugs in a computer program**

How then should we test?

How and what should we test?

This has been the subject of least 70 years of academic research…!

“Checking a large routine” – by Alan Turing



Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Consider the analogy of checking an addition. If it is given as:

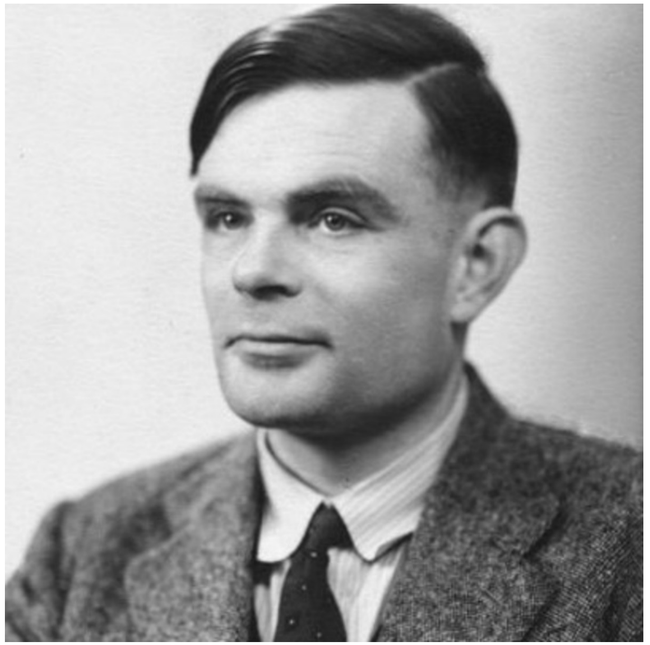
1374
5906
6719
4337
7768

26104

one must check the whole at one sitting, because of the carries.

But if the totals for the various columns are given, as below:

1374
5906
6719
4337
7768



How and what should we test?

There are many strategies and techniques.

I could not possibly cover them here.

Your company will likely have its own standards and practices, and metrics for measuring test quality.

However, here are some general examples of test cases…

The *Positive* Test Case

Key Idea

Establish some intended behaviour of the system happens as expected.

Process

Consult the specification of what the system should do, and design corresponding tests cases.

Examples

- Testing the system accepts correct, well-formed inputs
- Testing the system produces the right outputs, given those inputs

The *Negative* Test Case

Key Idea

Establish some unintended behaviour of the system does not happen.

Process

Considering the ways the system may crash or fail, e.g. with unexpected inputs or corner cases.

Examples

- Testing the system does not accept incorrect inputs
- Testing the system handles exceptional situations correctly, by failing gracefully, reporting problems to the user, etc.
- Testing the system is secure – testing to ensure that sensitive information is not revealed inadvertently

The *State Change* Test Case

Key Idea

Establish the system changes state correctly.

Process

An extension of the positive test case, but to ensure the internal state of the system is correct

Examples

- Testing the system logs in and logs out correctly.

The *Boundary* Test Case

Key Idea

Programmers typically make mistakes around the boundaries of conditions in a program.

Process

Throughly test the system around boundaries involved in inputs or key conditions.

Examples

- Given some condition `if (x > 0)`, test with `x = 0` and `x = 1`
  (the programmer may have meant to use “>=” rather than “>”)

The *Domain Knowledge* Test Case

Key Idea

Test using domain knowledge about the system.

Process

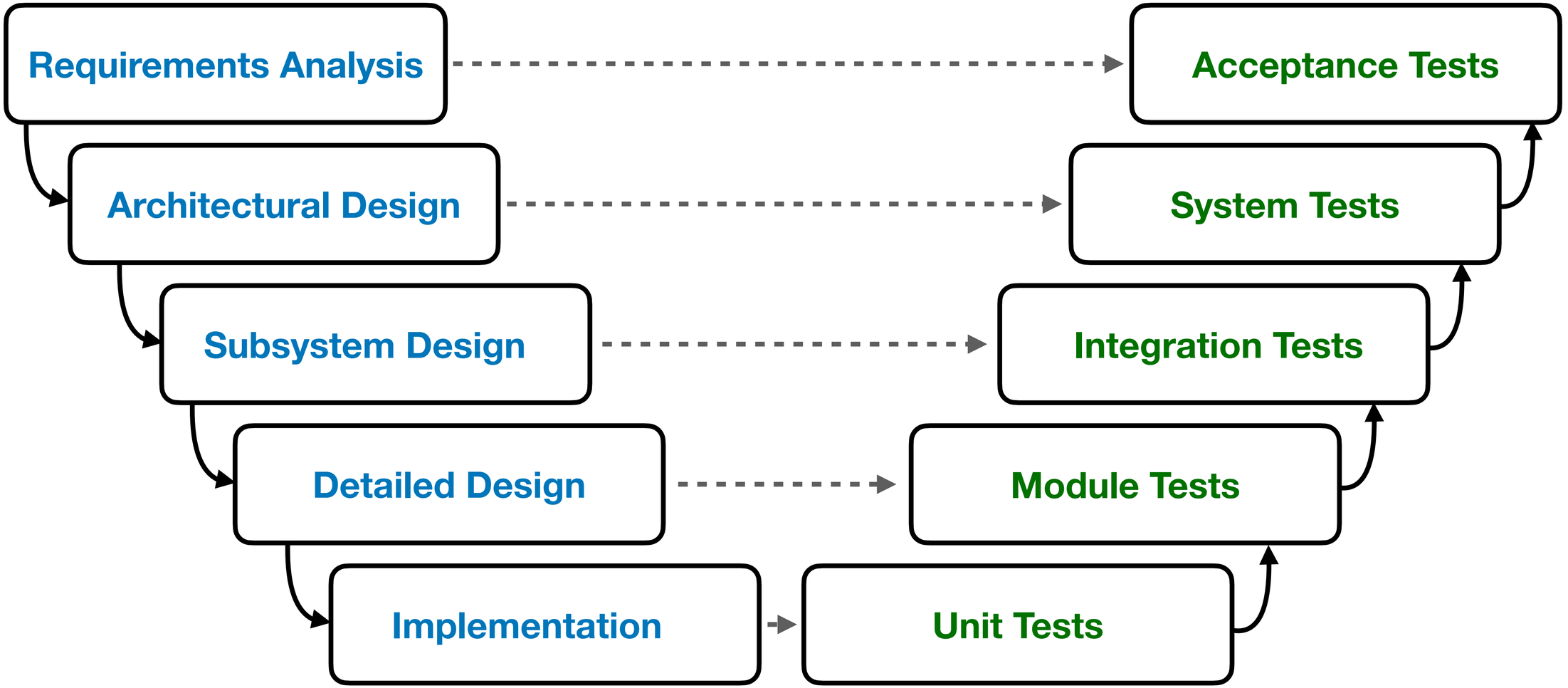
Trial and error thinking about certain inputs/scenarios that may cause it to fail.

Examples

- The date `29/2/2000` for a calendar system.

Types of Test

Types of testing – the “V-model”



Google's Categorisation of Tests

"Small" Tests. The simplest, restricted type of test. Must run in a single thread; cannot sleep; cannot perform I/O operations, like accessing the disk or network.

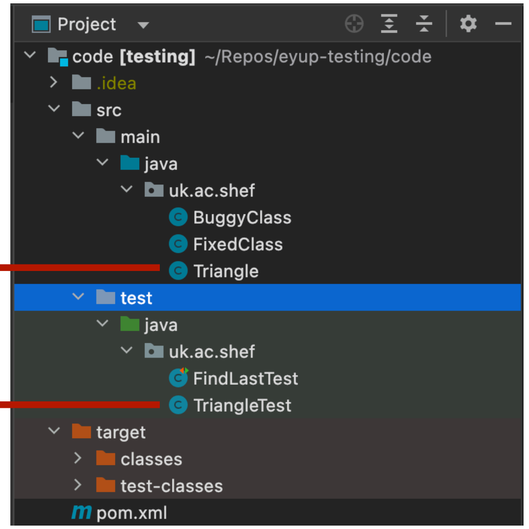
"Medium" Tests. Must run on a single machine. Can do all the above, but network calls must be restricted to the current machine.

"Big" Tests. All restrictions are removed.

Google advise **writing small tests as much as possible** to keep the entire test suite running *quickly* and *deterministically* (i.e., the same things happen all of the time, and the test suite is free of **flakiness**).

An Introduction to JUnit

Unit Testing in Java



Most project management tools like Maven keep tests and production code separate

Production code goes in `src/main/java`

Test code goes in `src/test/java`

We're going to test the `Triangle` class in a JUnit test class called `TriangleTest`

A JUnit Test Class and a Test

```java
import org.junit.jupiter.api.Test;


public class TriangleTest {

    @Test
    public void equilateralTest() {
      // Test code goes here
    }

    // ...
```

Tests are annotated with `@Test`
JUnit then knows which methods are test methods and which are helper methods

The Ingredients of a Test

```java
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class TriangleTest {

    @Test
    public void equilateralTest() {
        Triangle.Type result = Triangle.classify(10, 10, 10);
        assertEquals(Triangle.Type.EQUILATERAL, result);
    }

    // ...
```

We start by making method call(s) to set up the test and to the part of the system we want to test.

The Ingredients of a Test

```java
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class TriangleTest {

    @Test
    public void equilateralTest() {
        Triangle.Type result = Triangle.classify(10, 10, 10);
        assertEquals(Triangle.Type.EQUILATERAL, result);
    }

    // ...
```

We then write assertion statements to check the **actual result** is the one we **expected**.

Assertions

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class TriangleTest {

    @Test
    public void equilateralTest() {
        Triangle.Type result = Triangle.classify(10, 10, 10);
        assertEquals(Triangle.Type.EQUILATERAL, result);
    }

    // ...
```

The `assertEquals` method is a part of JUnit and specifically checks that some **expected value** is **equal** to the **actual one** returned from the unit being tested.

JUnit has a plethora of assertion types for checking relationships between actual and expected outputs.

These include `assertTrue(booleanVariable)`, `assertNull(reference)`, assertions on arrays and more. See:

https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

Checking for Exceptions with assertThrows

```
@Test
public void exceptionThrownWithInvalidTriangle() {
    InvalidTriangleException exception = assertThrows(InvalidTriangleException.class, () -> {
        Triangle.classify(0, 0, 0);
    });

    assertEquals("(0, 0, 0) is not a valid triangle", exception.getMessage());
}
```

JUnit v. Hamcrest Assertions

The default supplied JUnit assertions have some deficiencies:

- **It's easy with the assertion method parameters to get expected and actual the wrong way round.** (I do it all the time!) It's not consequential, which is why it's easy to do, but it could confuse other programmers.
- A different style is required for differing expected-actual relationships – i.e. a different assertion method
- The different assertion methods available are somewhat limited
- It's difficult to customise error messages

For these reasons, some programmers prefer the Hamcrest style of assertion.

Hamcrest Assertions

```java
@Test
public void isocelesTest() {
    Triangle.Type result = Triangle.classify(5, 10, 10);
    assertThat(result, equalTo(Triangle.Type.ISOSCELES));
}
```

Every assertion uses the generic `assertThat` method

The general assertion format maps more closely to natural language, making it more obvious that it's the **actual result** of the unit under test that goes first in the parameter order.

The relationship between actual and expected results is specified by a **matcher**, in this case, `equalTo`.

Hamcrest Matchers

Hamcrest has a plethora of “matchers”, like `equalTo`.

They can help write assertions involving a variety of types, including:

- Strings – e.g., can check if a string contains a substring, ignore case etc.
- Collections – whether an element is in a collection; what a collection contains, ignoring order etc.
- See **http://hamcrest.org/JavaHamcrest/javadoc/2.2/org/hamcrest/Matchers.html**

If the appropriate matcher is not available it’s very easy to write your own.
See **https://www.baeldung.com/java-junit-hamcrest-guide**

Code Coverage

Code coverage is a metric of how **much of the production code was executed by the tests**.

We can measure code coverage with tools like **JaCoCo**, which have plugins to tools like Maven.

This is sometimes useful, as we can see what code was not executed, and what we may need to write extra tests for – i.e., what we missed.

But bear in mind:

- Code coverage is just what the tests executed, not the code that it actually checked with assertions. **That is, coverage doesn't measure what was *actually* tested!**
- Sometimes 100% code coverage is infeasible to obtain with unit tests, or just not worth it. **"Gaming" the metric is not usually a wise use of time.**

JaCoCo

```java
public static Type classify(int side1, int side2, int side3) {
    Type type;

    if (side1 > side2) {
        int temp = side1;
        side1 = side2;
        side2 = temp;
    }
    if (side1 > side3) {
        int temp = side1;
        side1 = side3;
        side3 = temp;
    }
    if (side2 > side3) {
        int temp = side2;
        side2 = side3;
        side3 = temp;
    }

    if (side1 + side2 <= side3) {
        throw new InvalidTriangleException("(" + side1 + ", " + side2 + ", " + side3 + ") is not a valid triangle");
    } else {
        type = Type.SCALENE;
        if (side1 == side2) {
            if (side2 == side3) {
                type = Type.EQUILATERAL;
            } else {
                type = Type.ISOSCELES;
            }
        } else if (side2 == side3) {
            type = Type.ISOSCELES;
        }
    }

    return type;
}
```

How do Software Failures Happen?

The First Actual Bug – 1947

Photo # NH 96566-KN (Color) First Computer "Bug", 1947

9/9

0800 Arctan started

1000 " stopped - arctan ✓ { 1.2700 9.037 847 025
9.037 846 795 correct

13" u/c (032) MP - MC ~~1.9821447000~~ ~~2.1304776415 (033)~~ 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 11.00 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multy Adder Test.

1545 Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

~~1700~~ 1630 Arctangent started.

1700 closed down.

Relay 3145

Relay 3370

How do software failures happen?

1. The program location containing a **defect** is **reached** during execution.

2. The **defect infects** the state of the program

3. The infection **propagates** to the program's output causing a **failure**.

How do software failures happen?

1. The program location containing a **defect** is reached during execution.

2. The defect **infects** the state of the program

3. The infection propagates to the program's output causing a **failure**.

Defect

Infection

Failure

How do software failures happen?

1. The program location containing a defect is **reached** during execution.

2. The defect **infects** the state of the program

3. The infection **propagates** to the program's output causing a failure.

Reachability

Infection

Propagation

How do software failures happen?

Reachability

Infection

Propagation

Can you spot the defect?

```
public static int findLast(int[] x, int y) {
    for (int i = x.length - 1; i > 0; i--) {
        if (x[i] == y) {
            return i;
        }
    }
    return -1;
}
```

When does it **infect** the program?

When does it cause the program to produce a **failure**?

A test that exposes the failure… (and fails itself)

```
@Test
public void findLast() {
    int[] x = {1, 0};

    assertThat(BuggyClass.findLast(x, 1), equalTo(0));
}
```

The assertion is key. If we don’t check enough of the faulty method’s output, we could miss the failure!

A successful test must also reveal the failure.

Reachability

Infection

Propagation

Revealability

Reachability

Infection

Propagation

Revealability

Practical

https://github.com/philmcminn/eyup-testing

Practical

Navigate to the `uk.ac.shef` package (in the `code` directory of this repository) and open the `BuggyClass.java` class.

The class contains three methods. Each method has a defect, and you will need to write JUnit tests to establish the presence of the defect (by producing a failure) and that, following fixing the defect, the method now works.

For each method:

(a) Identify the defect, and write a JUnit test that causes it to fail. (The JUnit test should fail too.)

(b) Write a JUnit test that *does not* execute the defect.

(c) Write a JUnit test that executes the defect but *does not* cause a failure.

(d) Fix the defect and add it to `FixedClass.java`. Change the test you wrote in (a) to use the fixed method, and verify that it now passes.

Add your tests for each method into a new JUnit test class.