University of Sheffield

COM3529 Software Testing and Analysis

Introduction to Software Testing

Professor Phil McMinn

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

Level 0: Testing is the same as debugging

The basic, least mature view of testing is that of **Level 0 – testing is the same as debugging.**

At Level 0 thinking, programmers get their programs to compile, then debug the programs with a few arbitrary inputs.

This view does not distinguish between a program's **incorrect behaviour** and a **mistake** within the program. It also does very little to help develop software that is reliable or safe.

(… and testing is **not the same** as debugging, as demonstrated later.)

Level 1:
The purpose of software testing is to show software works

A significant step up from the naive Level 0.

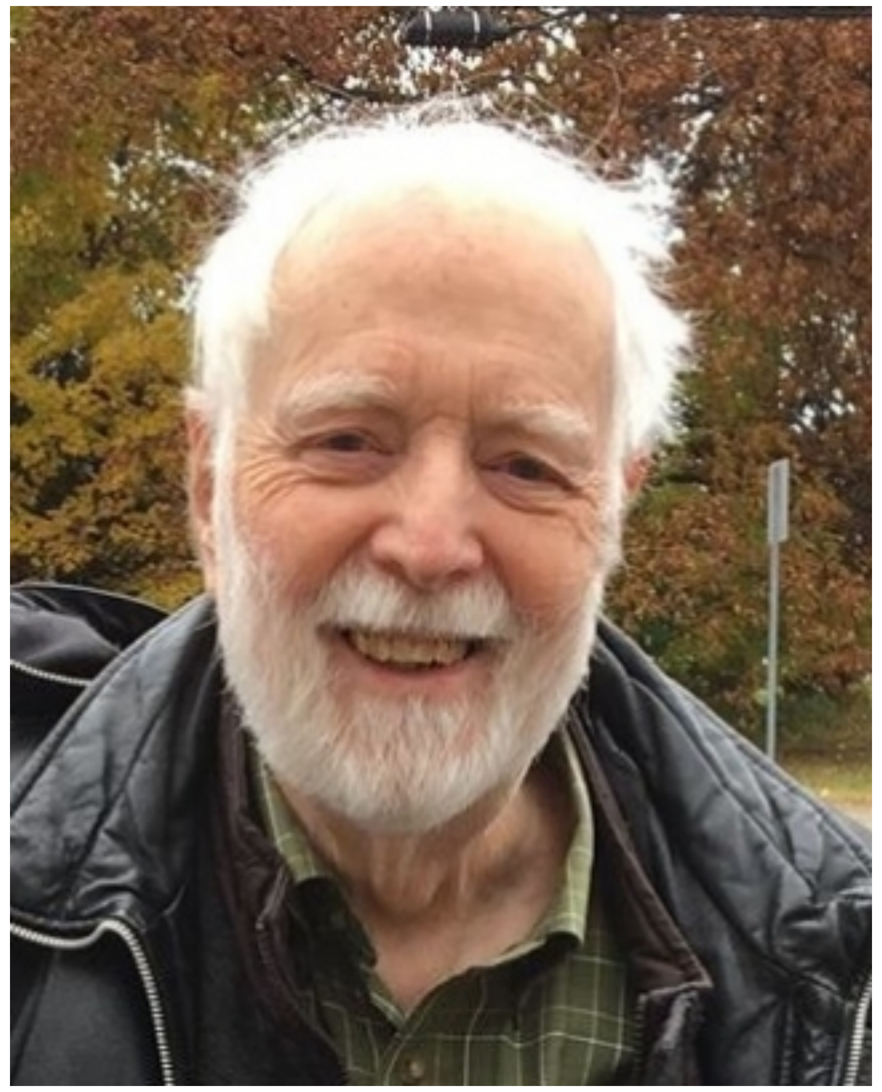
But, in any but the most trivial of programs, correctness is virtually impossible to either achieve or demonstrate!

(… another point we will return to later in the lecture.)

Level 2: The purpose of software testing is to show software *doesn't* work

Although looking for failures is certainly a valid goal, it is also a negative goal.

If a company is organised where testers and developers are on different teams, you may have a situation where testers may enjoy finding problems, but the developers never want to find problems – they want the software to work!



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

— Bill McKeeman

Level 2 testing puts testers and developers into an adversarial relationship, which can be bad for team morale.

Beyond that, when our primary goal is to look for failures, we are still left wondering what to do if no failures are found:

Is our work done?

and if so…

Is our software very good *or* **is our testing poor?**

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

Level 3 thinking lets us accept the fact that whenever we use software, we incur some risk.

Level 3 thinking lets us accept the fact that whenever we use software, we incur some risk.

The risk may be small, and the consequences unimportant, or the risk may be great and the consequences catastrophic, but risk is always there.

This allows us to realise that the entire development team wants the same thing – to reduce the risk of using the software.

In Level 3 testing, testing and developing go hand to hand to reduce risk.

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

Once the testers and developers are on the same “team”, or, testing is regarded as equally or even more important to development, an organisation can progress to **Level 4** testing.

Level 4 testing defines **testing as a mental discipline that increases quality**.

that helps all IT professionals develop higher quality software

Once the testers and developers are on the same “team”, or, testing is regarded as equally or even more important to development, an organisation can progress to **Level 4** testing.

Level 4 testing defines **testing as a mental discipline that increases quality**.

In the same way, Level 4 testing means that the purpose of testing is to improve the ability of the developers to produce higher quality software.

The best use of a spellchecker is not just to find misspelled words, but to improve our ability to spell.

Every time the spell checker finds an incorrectly spelled word, we have the opportunity to learn how to spell the word correctly.

The spell checker is the “expert” on spelling quality.

Beizer's Maturity Model

0. The same activity as debugging
1. Purpose is to show software works
2. Purpose is to show software doesn't work
3. Purpose is to reduce the risk of using software
4. Purpose is to help all IT professionals engineer better software

Level 1:
The purpose of software testing is to show software works

A significant step up from the naive Level 0.

But, in any but the most trivial of programs, correctness is virtually impossible to either achieve or demonstrate!

Why?

Why Finding ALL Bugs is Impossible

or

Why Software Testing is Hard

The Problem of Exhaustive Testing

```java
public 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^{32x6} or 2^{192} (≈ 6 x 10^{57}) unique inputs to try!

Suppose each input takes ≈ 1 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 and Software Testing

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 uncomputable too.

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

— Edsger Dijkstra

	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**

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

a human being makes a **manual** judgment

e.g. an **assertion in JUnit**

In software testing an **oracle** is **something** or **someone** who can determine if a **software output is correct**

~~ORACLE~~

No, not the company!

But we don’t need every single input 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.

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**

How do Software Failures Happen?

A Method and its Tests

```java
public static Set<Character> duplicateLetters(String s) {
    // lower case the string and remove all characters that are not letters
    s = s.toLowerCase().replaceAll("[^a-z.]", "");

    // initialise the result set
    Set<Character> duplicates = new TreeSet<>();

    // iterate through the string
    for (int i = 0; i < s.length(); i++) {
        char si = s.charAt(i);

        // iterate through the rest of the string, checking for the same letter
        for (int j = i; j < s.length(); j++) {
            char sj = s.charAt(j);

            if (si == sj) {
                // a match has been found, add it to the result set
                duplicates.add(si);
            }
        }
    }

    return duplicates;
}
```

A Method and its Tests

```
public class StringUtilsTest {

    @Test
    public void shouldReturnRepeatedChar() {
        Set<Character> resultSet = duplicateLetters("software testing");
        assertTrue(resultSet.contains('t'));
    }

    @Test
    public void shouldNotReturnNonRepeatedChar() {
        Set<Character> resultSet = duplicateLetters("software debugging");
        assertFalse(resultSet.contains('t'));
    }
}
```

PASSED

FAILED

```java
public static Set<Character> duplicateLetters(String s) {
    // lower case the string and remove all characters that are not letters
    s = s.toLowerCase().replaceAll("[^a-z.]", "");

    // initialise the result set
    Set<Character> duplicates = new TreeSet<>();

    // iterate through the string
    for (int i = 0; i < s.length(); i++) {
        char si = s.charAt(i);

        // iterate through the rest of the string, checking for the same letter
        for (int j = i; j < s.length(); j++) {
            char sj = s.charAt(j);

            if (si == sj) {
                // a match has been found, add it to the result set
                duplicates.add(si);
            }
        }
    }

    return duplicates;
}
```

This method contains a bug. Or more precisely, a **defect**.

Where is the defect? How does it cause the method to fail?

Defects

Software failures always start with the execution of one or more defects in the code.

A **defect** is simply a piece of **faulty, incorrect code**.

A defect may be a part of or a complete program statement, or may correspond to statements that *don't exist that should exist*.

Although programmers are responsible for making defects in the code, they may not be technically always be at fault – the problem may have arisen, for example, from a poorly specified set of requirements.

```java
public static Set<Character> duplicateLetters(String s) {
    // lower case the string and remove all characters that are not letters
    s = s.toLowerCase().replaceAll("[^a-z.]", "");

    // initialise the result set
    Set<Character> duplicates = new TreeSet<>();

    // iterate through the string
    for (int i = 0; i < s.length(); i++) {
        char si = s.charAt(i);

        // iterate through the rest of the string, checking for the same letter
        for (int j = i; j < s.length(); j++) {
            char sj = s.charAt(j);

            if (si == sj) {
                // a match has been found, add it to the result set
                duplicates.add(si);
            }
        }
    }

    return duplicates;
}
```

The defect in our example is with the second loop initialiser.

It should start iterating at `i + 1` to check for duplicates of the character at `i`, not at `i` itself (which is guaranteed to be identical!)

Infections

An **infection** is what happens when the defect is executed, and the program's state is affected.

When a program's state is infected it starts to work incorrectly:

- Variables start to take on the wrong values
- Decisions made in the program are evaluated incorrectly, and the execution path deviates from the correct one.

But at this point, it has not affected the output of the program (and the fault, so far, has had no observable effect).

```java
public static Set<Character> duplicateLetters(String s) {
    // lower case the string and remove all characters that are not letters
    s = s.toLowerCase().replaceAll("[^a-z.]", "");

    // initialise the result set
    Set<Character> duplicates = new TreeSet<>();

    // iterate through the string
    for (int i = 0; i < s.length(); i++) {
        char si = s.charAt(i);

        // iterate through the rest of the string, checking for the same letter
        for (int j = i; j < s.length(); j++) {
            char sj = s.charAt(j);

            if (si == sj) {
                // a match has been found, add it to the result set
                duplicates.add(si);
            }
        }
    }

    return duplicates;
}
```

The infection in our example causes the loop starts iterating an index in the string too early.

This further causes each character in the string to be added to the `duplicates` set. But at this point, there is nothing observably wrong with the program.

Failures

A **failure** occurs when the infection propagates to the output of the program.

That is, the program is observably behaving incorrectly.

```java
public static Set<Character> duplicateLetters(String s) {
    // lower case the string and remove all characters that are not letters
    s = s.toLowerCase().replaceAll("[^a-z.]", "");

    // initialise the result set
    Set<Character> duplicates = new TreeSet<>();

    // iterate through the string
    for (int i = 0; i < s.length(); i++) {
        char si = s.charAt(i);

        // iterate through the rest of the string, checking for the same letter
        for (int j = i; j < s.length(); j++) {
            char sj = s.charAt(j);

            if (si == sj) {
                // a match has been found, add it to the result set
                duplicates.add(si);
            }
        }
    }

    return duplicates;
}
```

Failures therefore depend on when programs deliver observable outputs. In our example, we interrogate the return value of the method in our tests, causing a **test failure**.

But during execution of the whole software, the method may be used internally by another method and a failure may happen much later, in a different place.

Failures vs Test Failures

We therefore need to distinguish between failures and test failures.

Failures are when software behaves incorrectly when run as a whole in production.

Test failures are when tests themselves fail because either:

(a) the test revealed a software failure

(b) the test itself was incorrect, for example it made an incorrect assertion about the behaviour of the software.

Testing vs Debugging

We can also now debunk the idea that testing and debugging are the same.

Testing is the process of evaluating software by observing its execution.

Debugging is the process of tracking failures/test failures back to the defects that were ultimately responsible for them.

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

Defects are not always reached (executed)

```
public static Set<Character> duplicateLetters(String s) {
    // lower case the string and remove all characters that are not letters
    s = s.toLowerCase().replaceAll("[^a-z.]", "");

    // initialise the result set
    Set<Character> duplicates = new TreeSet<>();

    // iterate through the string
    for (int i = 0; i < s.length(); i++) {
        char si = s.charAt(i);

        // iterate through the rest of the string, checking for the same letter
        for (int j = i; j < s.length(); j++) {
            char sj = s.charAt(j);

            if (si == sj) {
                // a match has been found, add it to the result set
                duplicates.add(si);
            }
        }
    }

    return duplicates;
}
```

Consider what happens if the inputs string s is empty.

A good test suite needs to exercise as much of the software as possible.

We will come back to this in later in the module.

Defects may not always cause infections

```java
public static Set<Character> duplicateLetters(String s) {
    // lower case the string and remove all characters that are not letters
    s = s.toLowerCase().replaceAll("[^a-z.]", "");

    // initialise the result set
    Set<Character> duplicates = new TreeSet<>();

    // iterate through the string
    for (int i = 0; i > s.length(); i++) {
        char si = s.charAt(i);

        // iterate through the rest of the string, checking for the same letter
        for (int j = i + 1; j < s.length(); j++) {
            char sj = s.charAt(j);

            if (si == sj) {
                // a match has been found, add it to the result set
                duplicates.add(si);
            }
        }
    }

    return duplicates;
}
```

Consider a defect where the loop test is reversed (i.e. “>” is used instead of “<”)

For empty strings, the defect would be executed, but no variables take on the wrong values and the loop body is *not* executed – as normal.

So for this specific input, there is no infection.

```java
public static Set<Character> duplicateLetters(String s) {
    // lower case the string and remove all characters that are not letters
    s = s.toLowerCase().replaceAll("[^a-z.]", "");

    // initialise the result set
    Set<Character> duplicates = new TreeSet<>();

    // iterate through the string
    for (int i = 0; i < s.length(); i++) {
        char si = s.charAt(i);

        // iterate through the rest of the string, checking for the same letter
        for (int j = i; j < s.length(); j++) {
            char sj = s.charAt(j);

            if (si == sj) {
                // a match has been found, add it to the result set
                duplicates.add(si);
            }
        }
    }

    return duplicates;
}
```

Infections may not always propagate to the output

Consider what happens with the original defect if the inputs string `s="stst"`.

The defect is executed and the characters "`s`" and "`t`" are entered into the `duplicates` set too early, but the overall output is correct.

Test cases need to reveal failures

The method fails with both tests, but only one of the tests “revealed” the failure (by causing a test failure).

```java
public class StringUtilsTest {

    @Test
    public void shouldReturnRepeatedChar() {
        Set<Character> resultSet = duplicateLetters("software testing");
        assertTrue(resultSet.contains('t'));
    }

    @Test
    public void shouldNotReturnNonRepeatedChar() {
        Set<Character> resultSet = duplicateLetters("software debugging");
        assertFalse(resultSet.contains('t'));
    }
}
```

PASSED

FAILED

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

Defect **R**eached 😫

State **I**nfected 🤢

Infection **P**ropagated 🤮

Failure **R**evealed 🤓

Roadmap of this Module

What we will cover – Weeks 1-5

Theoretical foundations (this lecture)

Types of testing

- Automated and Manual
- Unit, Integration, and System

Unit Testing

- Good practices
- Test doubles

Code Coverage

- Structural
- Logic
- Data Flow
- Input Domain

What we *won't* cover…

On the whole, we **won't** be covering specific technologies and environments.

The practices and techniques we will cover apply to any software system you are developing.

The course is designed to serve as the foundation of any testing that you need to do.

Each domain has its own testing practices and tools, however, so you will need to look for additional resources that focus on those when the time comes.

Assessment and Feedback

This module is solely assessed by an exam.

Preparation will be via the lab classes, where you will study problems similar to those that will appear on the exam.

We will provide feedback individually and via model answers.

Computer Laboratory Classes

We will set problem sheets and practical computer-based exercises that are designed to help you:

- **Understand the concepts explained in lectures**
- **Understand how the lecture material can be applied practically**
- **Prepare for the exam**

You can ask us any questions about the materials.

Teaching Team

Phil McMinn
Module Lecturer
Weeks 1-5

José Rojas
Module Lecturer
Weeks 6-10

Islam Elgendy
Teaching Assistant
Practical Sessions

Megan Maton
Demonstrator
Practical Sessions

Ruizhen (Rui) Gu
Demonstrator
Practical Sessions

Additional Notes / Follow-Up

The **Post Office software scandal** mentioned at the start of the lecture was the subject of an episode of the **BBC's "Panorama"**, which may still be available here:

https://www.bbc.co.uk/iplayer/episode/m0016t20/panorama-the-post-office-scandal