# CITS5501 Software Testing and Quality Assurance
# Introduction

Unit coordinator: Arran Stewart

# Introduction

# Highlights

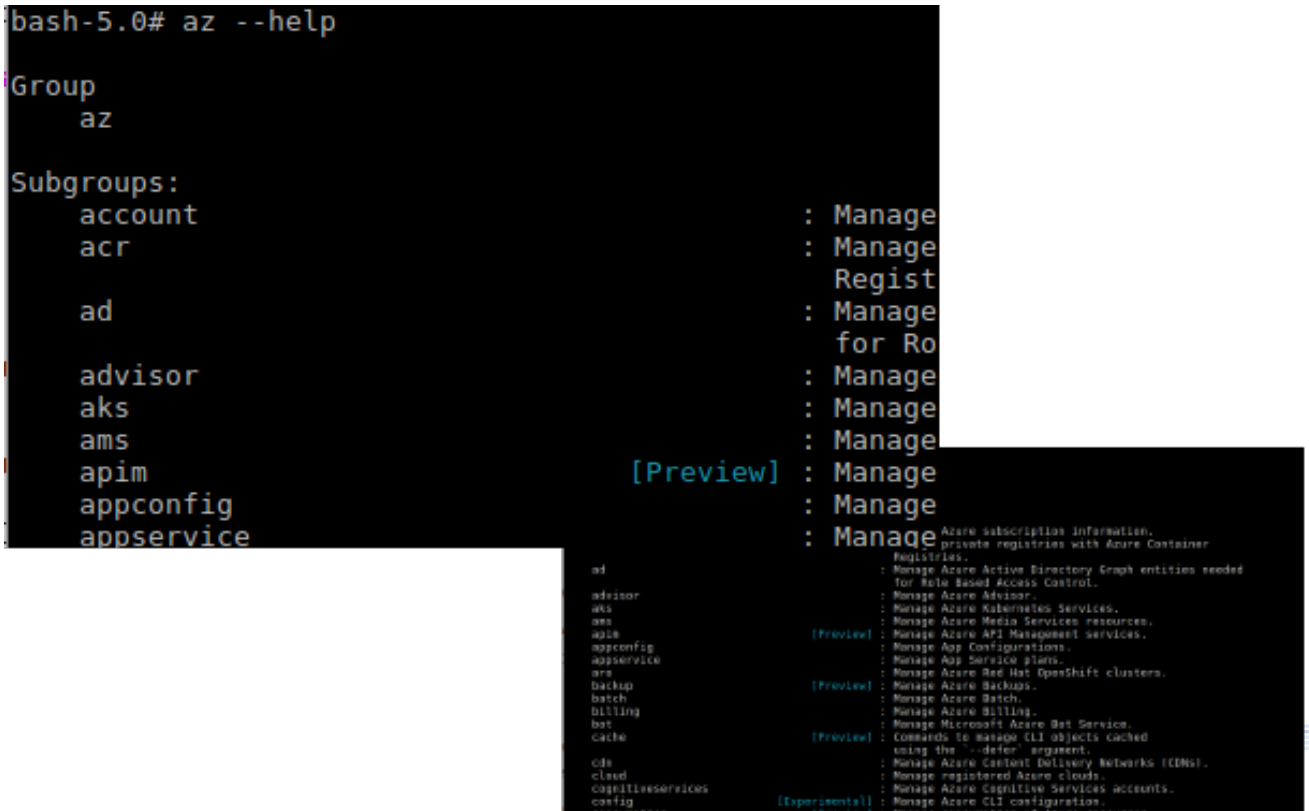
This lecture gives a big picture view of what we will cover and why.

The big questions –

- There is a huge diversity of software projects in existence – from web sites and apps, to systems embedded in hardware (anything from aeroplane sensors to washing machines), from tiny personal projects to programs running on supercomputers – how can we know how to test them and ensure they're of reasonable quality?
- For all these sorts of software projects – what makes them high (or low) quality? And how can we repeatedly ensure we produce software of high quality?

# Examples

- Just the `--help` message alone, for a medium-complexity program like those used to manage Amazon or Google or Azure cloud virtual machines, will typically show dozens or even hundreds of sub-commands, each with many options:



# Examples

- How would we go about testing that an application like this does what it says it does?
- Even more complex command-line applications include compilers (like `javac`, the Java compiler) – the specification alone for programs like this often runs to hundreds of pahes.

# Why are testing techniques useful?

- Some developers will be working on entirely novel projects, but often, we will be working with *legacy* software.
- If we are asked to make a change (a bug fix or improvement) to existing software – how do we know what we are doing is correct? How do we know we aren't introducing new bugs?
- When working with legacy software, often the first step is to ensure a good testing framework exists – otherwise, we potentially have no idea if our change has actually improved things, or made things worse.

- (Testing is important for novel, non-legacy software too, of course – but often the developers have a better understanding of what effects their changes are likely to have.)

# Types of testing

We will look at a wide range of testing and QA *techniques* – from the very simple, like unit testing (which *every* developer should be using), through to the technical and complex (formal methods and software modelling).

# Examples

Some examples of these sorts of techniques in use:

- Verification of software properties (e.g. the provably secure seL4 Microkernel)
- Model checking – Microsoft uses model checking techniques to test that driver code (which runs with high privileges) is using the API correctly
- Enforcing properties with rich type systems:
  - Memory safety
  - Microsoft's research Singularity OS
  - Encoding protocols using types ( session types )
- Extracting programs from proofs (using e.g. proof assistants like Agda)

# Methodology

In addition to various testing and QA techniques, we'll look at a general *methodology* for testing software.

Meaning that even when presented with a software system that is totally novel to you, or tools you've never used before, you'll still be able to design and implement an adequate testing and quality assurance plan.

# Admin

# Unit Information

Unit Coordinator: Arran Stewart
Contact: arran.stewart@uwa.edu.au
Phone: +61 8 6488 1945
Office: Rm G.08 CSSE Building
Consultation: Drop in from 4–5pm Wednesdays, or email for an appointment.

Unit webpage: accessible via GitHub, at
https://github.com/cits5501

# Announcements

Announcements will be made in lectures, and on the unit help forum, help5501.

It's important to check the forum regularly – at least once a week.

# Unit contact hours – details

Lectures:

- You should attend one lecture per week – you should either attend in person, attend online (we will use MS Teams), or watch the recorded lecture. (Recorded lectures are available via the university's LMS, at https://lms.uwa.edu.au/.)

Workshops:

- You should attend one lab/workshop each week, starting in week *two*.
  If there is room available for you, you are welcome to attend other lab sessions as well.
- In the lab/workshops, we will work through practical exercises related to the unit material. If you have a laptop, it may be useful to bring it, but you can use lab computers if not.

# Non-timetabled hours

A six-point unit is deemed to be equivalent to one quarter of a full-time workload, so you are expected to commit 10–12 hours per week to the unit, averaged over the entire semester.

Outside of the contact hours (3 hours per week) for the unit, the remainder of your time should be spent reading the recommended reading, attempting exercises and working on assignment tasks.

# Textbook

See the unit website for details of the textbooks you will need access to:

https://cits5501.github.io/

# Assessment

The assessment for CITS5501 consists of two short assessments (online quiz or an on-paper exercise), a project, and a final examination: see the Assessment page at

- https://cits5501.github.io/assessment/

In general, the short assessments should be pretty straightforward – do-able in an afternoon, at most – but we will allow most of a week from when they're available to when they're due.

The project will be done individually, and involves designing and executing a testing and validation strategy for some hypothetical software.

# Schedule

- General overview of topics:
  - Testing & testing methodology
  - Quality assurance
  - Formal methods and formal specifications
- The current unit schedule is available on the unit website:

  https://cits5501.github.io/schedule/
- The schedule below gives recommended readings for each topic: either chapters from the recommended texts, or extracts. Your understanding of the lecture and workshop material will be greatly enhanced if you work through these readings prior to attending.

# Prerequisites

The prerequisites for this unit are 12 points of programming units. At UWA, that should mean you're familiar with at least one object-oriented programming language (Java or Python).

If you aren't – let me know.

# Programming languages

We will mostly be using the Java programming language.

A *detailed* knowledge of Java is not essential – if you have a good knowledge of Python, instead, it should be straightforward to pick up the parts of the language you need.

(As a start – start experimenting with the `mypy` optional static type checker for Python.)

If you *aren't* familiar with Java, and would like some assistance getting familiar with the language, please post in the help forum (or send me an email, if you prefer) – I am putting together some suitable resources for beginners in the language.

# Programming languages

For one lecture, at the end of semester, we'll be using a language created by Microsoft Research for proving program correctness, Dafny.

It's more similar to the C# language than Java, but shouldn't be hard to pick up.

# Programming languages

We'll also be using a modelling language called *Alloy* (again, towards the end of semester).

It too has a syntax somewhat similar to Java.

# Software quality

# Software quality - what is it?

- What are some ways that software can be good?
  And what are some ways that it can be bad (or, less than ideal)?

# Ensuring quality software

- There are multiple aspects to building quality software:
  - Organisational Processes – How does the software team operate?
  - Process and Software Standards – Are particular standards used?
  - Process Improvement – How is success in building quality software measured and improved?
  - Requirements Specification – How do we work out what software we should be building? And how do we work out whether we built the right software?
  - Formal Methods – Ways of proving that software is correct
  - Testing, Testing, Testing – Identifying and correcting bugs

# The software “illities”

There are many features that contribute to the success of software, besides just it’s “correctness”:

- Usability
- Maintainability
- Scalability
- Reliability/Availability
- Extensibility
- Securitability [sic]
- Portability

# Types of testing

Testing is used in several ways in modern software development:

- Unit tests – Ensuring functional units are correct
- Integration testing – Ensuring components work together
- Acceptance testing – Getting paid at the end of the day
- Regression Testing – Don't break the build!
- Test Driven Design – "Test-first" software process
- Tests as documentation – Complete test suites are often the most accurate documentation a project has.

# Testing concepts

# Software Faults, Errors & Failures

- Software Fault: A static defect in the software
- Software Failure: External, incorrect behavior with respect to the **requirements** (or other description of the expected behavior)
- Software Error: An incorrect internal state that is the manifestation of some fault

# Software Faults, Errors & Failures

What are requirements?

Kinds of requirement or specification:

- Business needs ("why?")
- Requirements ("what?")
- System specifications ("how?")

In this unit, we will usually care less about what sort of requirement of specification something is, and more about the fact that we have to satisfy it.

# Fault and Failure Example

- A patient gives a doctor a list of symptoms
  (Failures)
- The doctor tries to diagnose the root cause, the ailment
  (Fault)
- The doctor may look for anomalous internal conditions (high blood pressure, irregular heartbeat, bacteria in the blood stream)
  (Errors – incorrect internal state)

# Goals of testing

Based on process maturity:

- Level 0: There's no difference between testing and debugging
- Level 1: The purpose of testing is to show correctness
- Level 2: The purpose of testing is to show that the software doesn't work
- Level 3: The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Level 4: Testing is a mental discipline that helps all IT professionals develop higher quality software

# Level 0 Thinking

- Testing is the same as debugging
- Does not distinguish between incorrect behavior and mistakes in the program
- Does not help develop software that is reliable or safe

# Level 1 Thinking

- Purpose is to show correctness
- Correctness is impossible to achieve
- What do we know if no failures?
  - Good software, or bad tests?
- Test engineers have no:
  - Strict goal
  - Real stopping rule
  - Formal test technique
  - Test managers are powerless

# Level 2 Thinking

- Purpose is to show failures
- Looking for failures is a negative activity
- Puts testers and developers into an adversarial relationship
- What if there are no failures?

# Level 3 Thinking

- Testing can only show the presence of failures
- Whenever we use software, we incur some risk
- Risk may be small and consequences unimportant
- Risk may be great and consequences catastrophic
- Testers and developers cooperate to reduce risk

# Level 4 Thinking

A mental discipline that increases quality

- Testing is only one way to increase quality
- Test engineers can become technical leaders of the project
- Primary responsibility to measure and improve software quality
- Their expertise should help the developers