CITS5501 Software Testing and Quality Assurance
Introduction

Unit coordinator: Arran Stewart

Introduction

Overview

- Goals
  - What is this course about?
  - What do we cover, and why?
- Admin
  - Unit website & announcements
  - Teaching activities
  - Assessment & feedback
  - Prerequisites
- Assessment tips
- Testing and QA introduction

Highlights

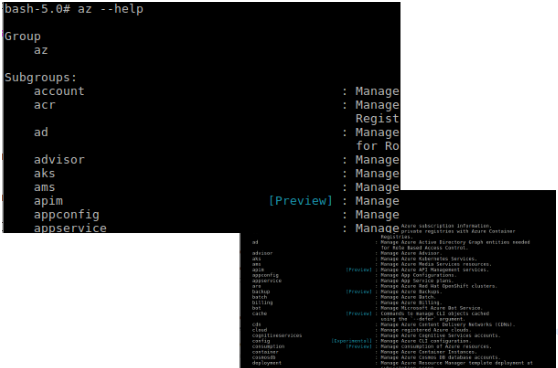
This lecture gives a big picture view of what we will cover and why.

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

- Software applications can be complex
- 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 pages.

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:

Data-driven testing
: used in JUnit and many other testing frameworks

Property-based testing
: first introduced in the Haskell language, and adopted in many others

Verification of software properties
: e.g. the provably secure seL4 Microkernel

Model checking
: Used e.g. by Microsoft to test that driver code (which runs with high privileges) is using the API correctly

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 coordinator

Unit Coordinator:	Arran Stewart
Email:	cits5501-pmc@uwa.edu.au
Office:	Room G.08 CSSE Building
Consultation:	Between 2–4pm Thursdays, or email for an appointment.
Unit website:	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 twice a week.
- If you log in and visit the forum site, you can set it to alert you via email when new postings are made.

Unit contact hours – details

Lectures:

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

Unit contact hours – details

Labs:

- Labs are run on a “drop-in” basis.
- Each week, a lab worksheet will be available which you can work through at your own pace from home or on campus
  - (Plus, in some weeks, unassessed exercises on the CSSE Moodle server at https://quiz.jinhong.org)
- If you encounter any issues or have questions, feel free to drop in to one of the lab sessions to ask them (on a first-come, first-served basis)
- All timetabled lab sessions can be viewed via the UWA Timetable site – see https://cits5501.github.io/#weekly-activities

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 project tasks.

Textbooks

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

https://cits5501.github.io/schedule/#recommended-readings

(Or just go to https://cits5501.github.io/faq and search for "textbook".)

Assessment

The assessment for CITS5501 consists of an online quiz, a mid-semester test, a project, and a final examination: see the Assessment page at

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

All assessments are to be done individually – there is no group or pair work – and are submitted using Moodle.

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 gives recommended readings for each topic: either chapters from the recommended texts, or extracts. Your understanding of the lecture and lab 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 statically type-checked programming language (usually Java or C).

If you aren't – let me know ASAP.

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 C, instead, it should be straightforward to pick up the parts of the language you need.

I will review some of the basics of the language in class; you should make sure you have access to a textbook on Java (almost any will do) to bring yourself up to speed.

Programming languages

At the end of semester, we'll spend a week or two 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?

Ensuring quality software, cont'd

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
: Identifying and correcting bugs

The software "illities"

There are many features that contribute to the success of software, besides just its "correctness" – for example:

- usability
- maintainability
- scalability
- reliability/availability
- extensibility
- securitability [sic]
- portability

The software "illities"

- usability
- maintainability
- scalability
- reliability/availability
- extensibility
- securitability [sic]
- portability

You should know from previous units that these are called **non-functional system properties**.

- They describe not what a system (or program, or module, or other unit of software) does, in terms of inputs and outputs – (i.e., its behaviour, when modelled **as a function**)
- – but rather, the manner in which it does it (securely, portably, etc.)

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

End-to-end testing
: Ensuring user stories about the system work as expected, in a particular environment

Acceptance testing
: Ensuring contractual obligations are met

Types of testing, cont'd

Regression testing
: Running test suites to ensure old bugs are re-introduced

System testing
: Ensuring the system as a whole works, and performs as expected under particular workloads

Test driven design
: Improving test quality by adopting "test-first" software processes

Tests as documentation
: Complete test suites are often the most accurate documentation a project has.

Testing concepts

Faults and failures

Software bug (aka *fault* or *defect*)

A defect in a system's *static artifacts* – usually source code – which at runtime will (if executed or encountered) cause the system to behave **incorrectly**.

Example: at a particular line in our program, a `for` loop accesses elements of an array `arr` from index position 0, through to index position `arr.length`. This will go out of bounds, and (in Java) throw an exception, which is almost never the intended behaviour.

Often, the static artifact will be a file of source code, but could be a configuration file, a build file (e.g. `Makefile`), or similar.

Faults and failures, cont'd

Software failure

Externally observable incorrect behavior with respect to the **requirements** (or other description of the expected behavior) of some software system, module or other unit.

Example: Instead of operating as expected (say, sorting a file), a program when run displays a stack-trace and crashes.

A **defect** exists in the source code (or other static artifact) even when the system is not running; we can point it out, and (usually) predict what *will* happen at run-time when it is encountered.

In contrast, a **failure** is a type of behaviour that we observe at run-time; if the system is not currently executing, then by definition, it's not exhibiting any failures.

Faults and failures, cont'd

Invariants

We will see that failures are usually the results of a violation of a system's **invariants** – facts that the developers expect to be true, at various points in the program.

Example: Amara *expects* that an array index in the loop they are writing will always be within bounds of the array.

Example: Bernardo writes a binary search routine, and expects it will only ever be passed a *sorted* array. But in the current project, Carla has passed an unsorted array. As a consequence, Carla will likely observe failures when the code is run.

Types of invariants include loop invariants, class invariants, and preconditions.

Once an invariant is violated, the system is no longer in a sensible state – assumptions under which developers wrote their code are no longer true – and (for many systems) it's unsafe for the program to continue running. (Continuing to run could do more harm than good.)

Faults and failures, cont'd

So, what exactly happens if an invariant is violated?

If we are lucky, the violation might cause the program to quickly terminate

- (e.g. If Amara's loop is Java or Python code, an out-of-bounds exception will be thrown, and the program will likely terminate.)

If we are unlucky, the program will continue, but is likely to produce incorrect results later, or to corrupt in-memory data

- (e.g. If Amara's loop is written in C, no bounds checking occurs, but data will likely be silently corrupted.)
- (e.g. Carla's code that called Bernardo's binary-search will likely give incorrect results, but probably will not result in an exception.)

Sometimes a running system in which an invariant has been violated is said to be in an *erroneous* or *inconsistent* state.

Failures – incorrect behaviour

We said a failure is when some software does not behave in accordance with its *requirements*.

What are requirements?

Kinds of requirement or specification:

- Business needs ("why is this needed? what value does it have for the business?")
- Requirements ("what should the system do? and what performance criteria must it meet?")
- System specifications ("the requirements will be met by constructing subsystems and modules that satisfy criteria A, B, C, etc")

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

Fault and failure scenarios

What are the following – faults or failures?

- Daniel is writing his project report for CITS5501. He attempts to save the report, but instead, the report contents is corrupted.

Fault and failure scenarios

What are the following – faults or failures?

- Daniel is writing his project report for CITS5501. He attempts to save the report, but instead, the report contents is corrupted.
- In code she is writing, Elise calls the `Thread.sleep()` method in Java, which is used to suspend execution of a program thread of control.
  The method is supposed to be passed a number of milliseconds to sleep for, but Elise inadvertently passes the number of *micro*seconds.

Goals of testing

Some conceptions of the goals of testing – which do you think are correct?

- The purpose of testing is to show correctness of software
- The purpose of testing is to identify defects in the software
- The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Testing is a mental discipline that helps all IT professionals develop higher quality software