CITS5501 Software Testing and Quality Assurance
System, integration and regression testing

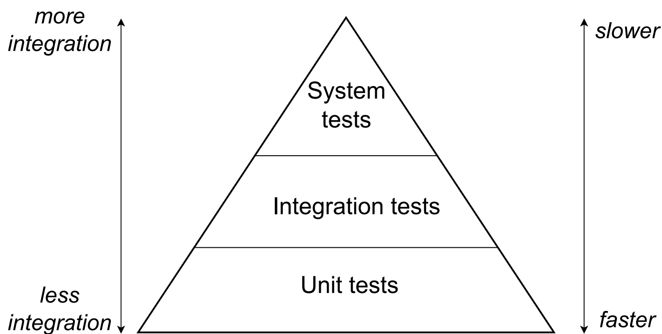
Unit coordinator: Arran Stewart

Overview

- Integration tests: what are they, why do them, what defects do they help find?
- Testing strategy
- System testing
- Other types of testing:
  - Regression testing
  - “Smoke” testing
  - End-to-end
  - Alpha/beta

Types of tests

Recall the "test pyramid" from lecture 3:



What is integration testing?

Recall that unit tests test some small "unit" of software

- Purpose: check the *behaviour* of that unit – exercise it and look for deviations from its specification
- We normally *mock* other external classes used in the test

Integration testing focuses on the flow of data and information between components, and their *interface*

- It asks, "Do the components work properly together?"
- The goal is to test the interfaces between components, and the interaction of components.

Why do integration testing?

- Unit tests only test some unit in isolation – not how it interacts with other units.
- Many failures arise from faults in the interaction between components
- We don't unit test third-party components we didn't write – but we *should* test how those components interact with others.
- Picks up problems we otherwise might not find til system testing (when they will be more expensive to fix)

Integration testing

- The entire system is viewed as a collection of components or subsystems (e.g. sets of classes).
- The order in which the components are selected for testing and integration determines the **testing strategy** – e.g. top-down, bottom-up (more on these later)

Examples of integration faults

What sort of defects might integration tests help identify?

We consider some examples.

Example

One component A tries to invokes another component (B), but misidentifies B.

In statically typed languages, the compiler will tell us if we're trying to invoke a non-existent class or method (e.g. `javac` will complain of undefined symbols).

But in dynamically typed languages like Python, or where we are e.g. invoking an HTTP API, this might not be detected until runtime.

Examples of integration faults

Example

One component A invokes another component (B), but does so incorrectly.

The developer of A might have misread the documentation for component B, or the API for B might have changed.

Component B might specify that calls have to happen in a particular order, and A fails to do this. (E.g. Often, C libraries require the library to initialized in some way before any other functions are called.)

Sometimes the use of *mocks* (test doubles) might pick this up, but work put into mocks does have diminishing returns – at some point it may be simpler to rely on integration tests.

Component A might not properly handle all the possible errors or exceptions thrown by B.

Examples of integration faults

Example

Components have inconsistent interpretation of parameters or values.

Can be seen as a sub-category of the previous example.

Component B might specify that a parameter should be in terms of particular units[1] (say, milliseconds or pounds per square inch), and component A assumes different units. (This was the cause of a failure in NASA's Mars Climate Orbiter.)

The parameters might be in a form such as XML or JSON (especially common for HTTP APIs), and the schema used to validate the parameters might have changed.

[1] When we discuss formal methods and type systems, we will look at ways of preventing this sort of error.

Examples of integration faults

Example

Two components have conflicting side effects.

For instance, both components might try to make use of the same file or database.

In the case of a file, this could lead to file corruption; in the case of a database (or an OS where open files are automatically locked), it could lead to deadlock or other concurrency issues.

Examples of integration faults

Example

One component A invokes another component (B), but communication is slow, unreliable, requires authentication, or depends on exact timing to be correct.

If a system is distributed over the network, or even just across different processes or threads, communication and concurrency problems may be identified during integration testing that aren't obvious when unit testing.

Examples of integration faults, cont'd

Example

Non-functional properties (e.g. performance, security) fail to hold when components are used together (**emergent failures**).

- We said that many qualities of a system (e.g. performance, security) can't be localised to a single component, but arise from the interaction of components.
- It follows that *failures* relating to those qualities (poor performance, poor security) sometimes can only be detected from the interaction of components

Testing strategy

- A common approach – begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
  - Start with units (functions/classes)
  - Then start integrating them

Testing strategy

- While doing unit testing, we will typically make use of "mocks"/doubles in place of other units or modules
- In integration testing, we can test how two (or more) units or modules work together
  - The units or modules under test will *not* be mocked, obviously, since what we want to know is whether they work properly together
  - But they might rely on additional components (e.g. databases) which *are* mocked.
- The closer we get to the top of the test pyramid, the fewer mocks we use, in favour of using components which are as close to the production environment as we can get.

This is an illustration of a "bottom-up" approach. We now compare several different integration approaches.

Integration testing strategies

Main options:

- Big bang integration (nonincremental)
- Bottom up integration
- Top down integration
- Sandwich testing
- Variations of the above

Drivers and stubs

Terminology sometimes used in integration testing:

- **Driver**: A program that makes calls into the module being tested and reports the results
  - The driver simulates some module that (in the final system) *will call* the module under test
- **Stub**: A module that has the same interface as the module under test, but is simpler
  - The stub simulates a module which is *called by* the module under test

“Big Bang” Integration Testing

The approach:

- Do no integration testing until all modules have been completed;
  then try and test everything at once.

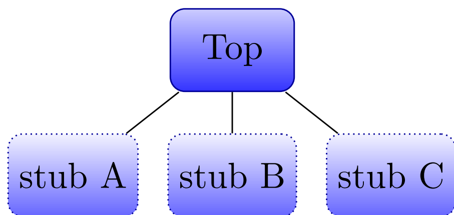
Problems:

- Expensive, if faults could've been detected earlier
- Poor ability to observe faults and diagnose/localize them

Top-down integration

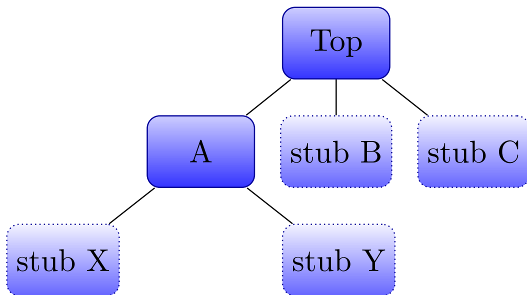
- Test the top layer or controlling subsystem first
  - It's the "top" module in the sense that it *uses* or calls into other modules
- Use stubs to simulate components we haven't implemented/integrated yet
- Then start implementing the subsystems called by that top system, and test them in the same way ...
- And continue "down" until everything is done.

Top-down integration



Begin with the top level, test it by letting it call stubs.
(From material earlier on test doubles: our stubs can be *spies*, that allow us check how they're being called and whether it's being done correctly.)

Top-down integration



As we implement and incorporate more modules, test *them* using stubs.

Pros and cons of top-down integration testing

Pro:

- Test cases can be defined in terms of the functionality of the system (functional requirements)

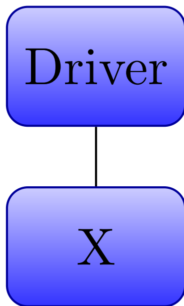
Cons:

- Writing stubs can be difficult: stubs should allow for a wide range of conditions to be tested.
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.
- One solution to avoid too many stubs: Modified top-down testing strategy
  - Test each layer of the system decomposition individually before merging the layers
  - Disadvantage of modified top-down testing: Both stubs and drivers are needed

Bottom-up integration

- Start by implementing and testing the modules/subsystems in the “lowest” layer, individually
- Use test drivers to simulate calling into them
- Then start replacing drivers with actual implementations, and work “upwards”

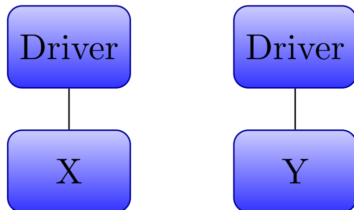
Bottom-up integration



Start by implementing modules at the *bottom* of the “uses” hierarchy.

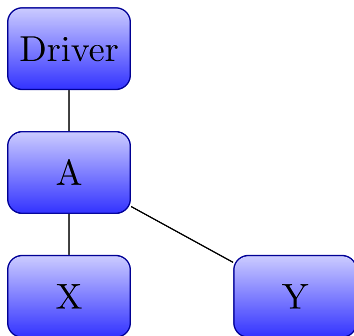
They will be tested by *drivers*, which simulate making calls into the module under test.

Bottom-up integration



As we implement more modules, we need to write drivers for them, too.

Bottom-up integration



But once we've finished a "mid-layer" module, it replaces the driver modules which previously simulated it.

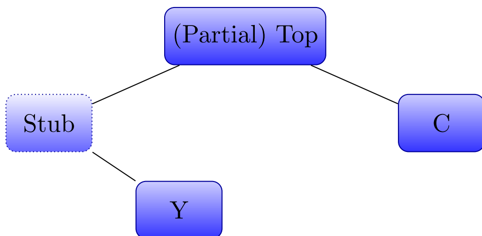
Pros and cons of bottom up integration testing

- Pro: Systems tested as they are ready
- Con: Typically tests one important subsystem (UI) last

“Sandwich” integration

- Combine top-down with bottom-up – work from both “ends” inwards

“Sandwich” integration



We may end up not needing as many stubs or drivers as in previous approaches.

Steps in integration testing

1. Based on the integration strategy, select a component to be tested. Unit test all the classes in the component.
2. Put selected component together; do any preliminary fix-up necessary to make the integration test operational (drivers, stubs)
3. Do functional testing: Define test cases that exercise all uses cases with the selected component
4. Do structural testing: Define test cases that exercise the selected component
5. Execute performance tests
6. Keep records of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

   The primary goal of integration testing is to identify errors in the (current) component configuration.

Which integration strategy should you use?

A useful question to ask is: what aspects or components of the system are most risky?

- We typically want to reduce risk by developing and testing (and/or prototyping) those early, so we can thoroughly test them and their interaction with other components
- And if there are parts of the system that are very *well* understood, we might leave them til later and/or start work in parallel

Other factors to consider:

- Scheduling constraints (we may know hardware or third-party components needed will not be available til a later date)
- Estimated amount of test harness code needed (stubs & drivers)

Which integration strategy should you use?, cont'd

Bottom up approach

- Good for object oriented design methodologies
- Test driver interfaces must match component interfaces
- Top-level components are usually important and cannot be neglected up to the end of testing
- Detection of design errors postponed until end of testing

Which integration strategy should you use?, cont'd

Top down approach

- Test cases can be defined in terms of functions examined
- Need to maintain correctness of test stubs
- Writing stubs can be difficult

Which integration strategy should you use?, cont'd

Sandwich

- Top and bottom layer tests can be done in parallel
- More flexible

System testing

System tests aim to test the entire system against its requirements and specifications.

Some categories of system tests:

- Tests of functional requirements
- Tests of non-functional requirements
- Acceptance tests (validates client expectations)

Tests of functional requirements

Sometimes called “functional testing”.

- Treats the whole system like a *function* or “black box” – so we can apply the same sorts of test design approaches (e.g. ISP) we have used for e.g. unit tests.
- By the time we do system testing, we may have additional system documentation (e.g. user manuals, online help) which can be used in test design

Tests of non-functional requirements

- Recall that many **non-functional** requirements (security, scalability, maintainability, performance) are *emergent* properties: they aren't a property of any single component in isolation, but rather emerge from the way multiple components interact as a whole
- This means that towards the *bottom* of the test pyramid, we'll often be more focussed on testing functional requirements/specifications
- But as we move towards the top of the pyramid, it becomes possible to test for non-functional requirements.

Tests of non-functional requirements

Some sorts of non-functional, system-level test:

- Load testing – How does our software perform under high loads – the largest volumes of data we expect to receive? Does it perform correctly?
- Stress testing – How does our software behave when we *exceed* the expected maximum?
  Does it degrade gracefully?
- Robustness testing – How well does our system handle malformed inputs?
  Does it avoid undesirable behaviours (e.g. segfaults, security holes, displaying raw stack traces to end users)?

Testing non-functional requirements

What do tests of this sort look like?

Good tests follow exactly the same pattern we've seen previously – Arrange, Act, Assert.

For a unit or integration test, we usually "Act" (invoke behaviour) by calling a method or function.

But for tests of non-functional requirements, we could be

- invoking the whole program and measuring particular properties (e.g. how long it takes to execute)
- starting a program (e.g. a web app), making requests against it, and measuring the response to those requests

Frameworks for non-functional testing

Sometimes we might write our tests of non-functional requirements in the same language(s) as our system, sometimes not.

Scripting languages like Bash, Python, and Perl are especially convenient for executing programs, launching other test utilities, and extracting performance data from the OS.

So even if our system is written in Java, it might be convenient to write these tests using a Perl or Python test framework.

Testing non-functional requirements

Besides load, stress, and robustness testing, some other sorts of system testing include:

- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - verify that the system meets security requirements and is protected from improper penetration
- Performance Testing
  - test the run-time performance of software within the context of an integrated system

Load, stress and robustness testing

For load and stress testing, we will normally generate random traffic/data for our system, and conduct tests which measure performance against requirements.

For robustness testing, *fuzzing* (and other sorts of randomized testing) can be very effective.

Security testing

Note that security problems cannot (typically) be avoided through testing alone – good system security requires us to be mindful of security and incorporate it at all stages of the software development lifecycle.

Types of security testing

Some typical sorts of security testing:

- Vulnerability scanning: using automated software which aims to detect known security vulnerabilities in a system.
  - Vulnerabilities detected can include misconfigured software, versions of particular packages known to be insecure, and more
  - The term "vulnerability scanner" normally means a program which is run against a live (running) system.
  - Some popular vulnerability scanners include Nessus and Nexpose (both commercial), or Nmap and Metasploit (partially or wholly open source)

Types of security testing, cont'd

- Penetration testing
  - This simulates an attack by a malicious party. It usually involves evaluating a system (including vulnerability scanning) and exploiting found vulnerabilities to gain access to the system and breach data confidentiality, data integrity, or the availability of services.
- Fuzzing
  - Fuzzing, which we've looked at previously, can often identify security vulnerabilities. The most common cause of program crashes is improper access to memory locations, and these can often be exploited so as to compromise security.

Secure software development techniques

Security tests should be part of a broader approach to security which might include:

- Threat modelling: a structured way of identifying threats and mitigations that could affect a system, and then organizing and communicating that information. (The OWASP page on threat modelling has more information on this.)
- Security reviews: review of code (or other artifacts, e.g. design documents or specifications) by a human reviewer, looking for insecure or problematic code.
- Static code analysis: using programs which analyse code statically (i.e., without running it), and aim to detect code that is likely to cause security problems or is known to be problematic in other ways.

Secure software development techniques, cont'd

- Compliance or conformance testing: assessing whether a system conforms to particular standards.
- Security audits: a type of security review; a security audit is a structured process for reviewing a system according to some defined standard.

Other sorts of testing

Other sorts of testing

We discuss some other sorts of testing you may encounter.

Regression testing

- Mentioned in previous lectures:
  Regression testing is the re-execution of some subset of tests that have already been conducted, to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated tools.

Smoke Testing

A common approach for creating "daily builds" for product software
Smoke testing steps:

- Software components that have been translated into code are integrated into a "build."
  - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
- A series of tests is designed to expose errors that will keep the build from properly performing its function.
  - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
- The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
  - The integration approach may be top down or bottom up.

Other sorts of testing

End-to-end testing

- Checks how a system or component behaves in a particular user-focused scenario (e.g. use case, or user story), usually in a near-production environment, and whether it behaves as expected.
- The focus differs a little from typical "system tests"
- System tests show that the system satisfies some requirement or specification
- End-to-end tests demonstrates that some particular task can be done by or using the system
- e.g. Can a user successfully login, go to the product page, add a product to the shopping cart, pay for items, and log out.

Other sorts of testing, cont'd

Validation testing

- Ensures that the product actually meets the client's needs
- Demonstrates that the system fulfills its intended use when deployed in an appropriate environment

Alpha and beta testing

- Focus is on customer usage
- Alpha testing = done by employees of development organisation, simulates typical use tasks
- Beta testing = done by releasing to a limited number of real users

Other sorts of testing, cont'd

Canary testing

- A new version of software, or new features, is initially released only to a small subset of users.[2]
- This might be a cohort who voluntarily elect to test new features, might be selected from e.g. a particular timezone, or might be selected at random.
- Ideally, a good testing regime will have already removed serious defects from the new software.
- But if they *have* got in, then exposing only a subset of users to problems can be preferable to exposing the entire user base.
- (Question: do you know of any system release problems which could have been avoided through canary testing?)

[2] Origin of the name: canaries were used in coal mines as an "early warning" to test for dangerous levels of odourless but toxic gases.

Other sorts of testing, cont'd

A/B testing

- A controlled experiment is done of two different versions of some software or interface – a control group ("A") and a "treatment" group using a novel version ("B")
- The aim is to obtain statistically rigorous results about which version performs better in some way.
- Example use: for websites with a very large user-base (e.g. Facebook). A change in interface might be tested using A/B testing to see if it is an improvement in some way over the existing interface.