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

Unit coordinator: Arran Stewart

Overview

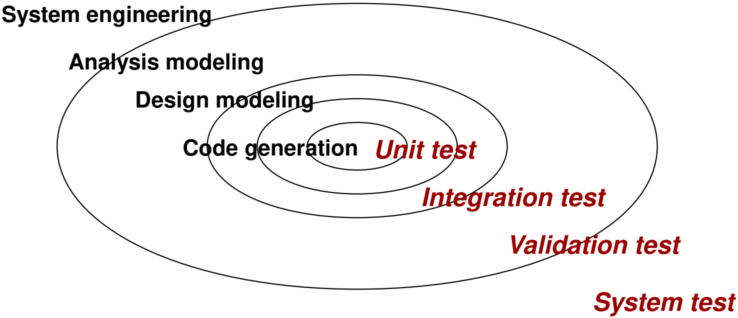
- Testing strategy
- Integration testing
- Regression testing
- “Smoke” testing
- Web testing

Types of tests

- We've looked in detail at unit tests, which test some "unit" of software
    - They are intended to check the *behaviour* of that unit – to 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 two components, and their *interface*
    - It asks, "Do they work properly together?"

Testing strategy

Types of testing:



Testing strategy

- Typically, we 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 units or modules work together

Integration vs unit testing

- Do we need both?

Integration vs unit testing

- Do we need both?

Yes . . .

- Unit testing is a necessary basis for integration testing
  - gives maximum control over individual units
- Integration testing
  - *may* discover module faults not found in unit testing – but that's a sign of insufficient unit testing
  - Ideally, should discover faults in the interfaces / flow of control between otherwise correct modules
  - Can be used to test third-party components which we can't unit test

Why do integration testing?

- Unit tests only test the unit in isolation
- Many failures arise from faults in the *interaction* between components
- Letting faults persist until system testing or deployment can be very expensive

Integration testing

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.
- The order in which the subsystems are selected for testing and integration determines the testing strategy

Examples of integration faults

- One component calls another incorrectly
  - e.g. perhaps calls must happen in a particular order
- Components have inconsistent interpretation of parameters or values
  - e.g. a parameter represents units of force – but is it in Newtons (SI system) or pounds (US)? (Cause of a Martian Lander fault)
- Conflicts arising due to side effects
  - e.g. two components try to make use of same temporary file

Examples of integration faults, cont”d

- Emergent faults (non-functional properties)
    - Many qualities of a system (e.g. performance, security) can’t be localised to a single component, but arise from the interaction of components.

Integration testing strategies

Main options:

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

Drivers and stubs

- **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.

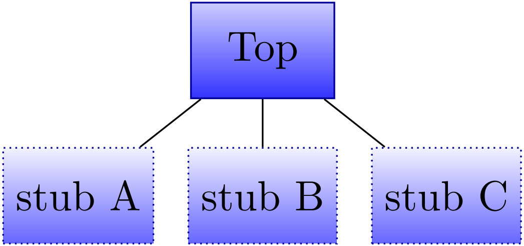
Problems:

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

Top-down integration

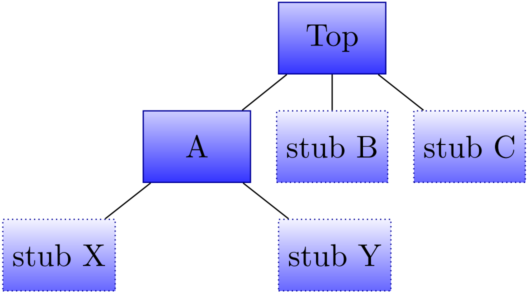
- Test the top layer or controlling subsystem first
  - It's the "top" module in the sense that it *uses* or calls into other modules
- Use stubs to simulate components we haven't implemented/integrated yet
- Then start implementing the subsystems called by that top system, and test them in the same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

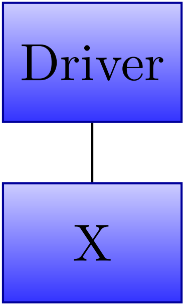
Cons:

- Writing stubs can be difficult: Stubs must allow all possible 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”

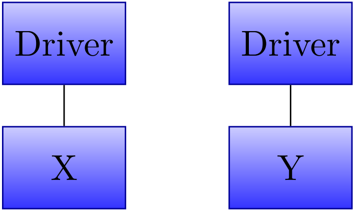
Bottom-up integration



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

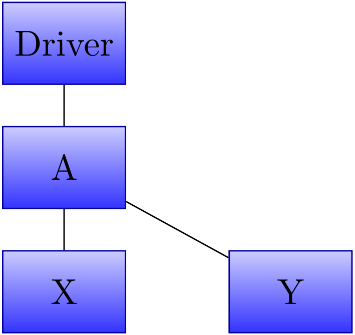
They will be tested by *drivers*, which simulate making calls into the module under test.

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.

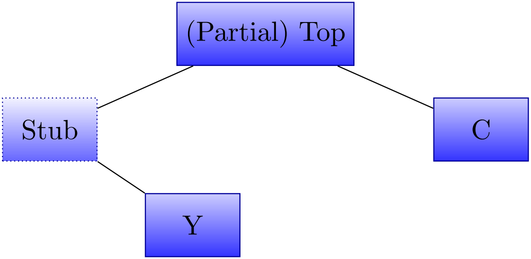
Pros and cons of bottom up integration testing

- 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?

- Factors to consider
  - Amount of test harness (stubs &drivers)
  - Location of critical parts in the system
  - Availability of hardware
  - Availability of components
  - Scheduling concerns

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

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 (due to testing or for other reasons) 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.

WebApp Testing - I

- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- Each functional component is unit tested.

WebApp Testing - II

- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end-users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

Other sorts of testing

- Validation testing
  - Focus is on software requirements
- System testing
  - Focus is on integration of sub-systems
- Alpha/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

- Recovery testing
    - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
    - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing
    - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
    - test the run-time performance of software within the context of an integrated system