Software Testing - An overview

- General concepts
- Formality of testing
- Test design
- Unit testing

Tests

- Tests relate to unit/component, integration, system and validation testing (see V-diagram)
- Each test class is different and requires its own techniques:
  - Unit/component test: to show whether a unit does/does not satisfy requirements and/or unit's implementation structure does/does not match design
  - Integration test: to show whether combinations of components are incorrect or inconsistent
  - System test: concerns issues and behaviour that can only be exposed by testing the entire system (performance, security, recovery, …)
  - Validation test/acceptance test: to show that the system meets the client's requirements - related to *use cases/scenarios*

Verification vs validation

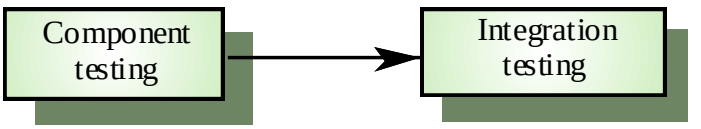
- One aspect of testing is aimed at answering the question: "Does the software do what the specification says it should?"
  - This is called "verification"
  - *"Are we building the system right?"*
- We will deal principally with verification
- A slightly different concept is "validation" testing, or "acceptance" testing
  - Making sure that the specification (and so the system that we build) is really what the client wants!
  - *"Are we building the right system?"*
- Validation requires carefully organized test plans, probably designed around the use cases/scenarios, and run through by the developer and client together
  - User acceptance tests may act as the final "quality gateway" and enable the client to "sign off" the contract

The impossibility of perfect testing

- The "ideal": *exhaustive testing:* try out all possible inputs
- For all but the simplest of programs, this is **not feasible**
  - There are just too many possibilities
  - Consider a program that does something based on two input integers…
    `int`s are roughly $-2*10^9$ to $+2*10^9$
    So number of tests is $4*10^9 * 4*10^9 = 16*10^{18}$
  - At one test per microsecond: $16*10^{12}$ seconds
  - This is roughly: ??? years
- **So in practice test data must be selected - *designed***
- And "*Program testing can be used to show the presence of bugs, but never to show their absence!*" (E W Dijkstra)
- Testing can do no more than increase our confidence
- A useful idea: A successful test is **one that detects an error**!

The testing process

- Component/unit testing
  - Testing of individual program components
  - Usually the responsibility of the component developer (except sometimes for critical systems)
  - Tests are derived from component descriptions/ specifications
- Integration testing
  - Testing of *groups* of components integrated to create a system or sub-system
  - Often the responsibility of an independent testing team
  - Tests are based on a system specification



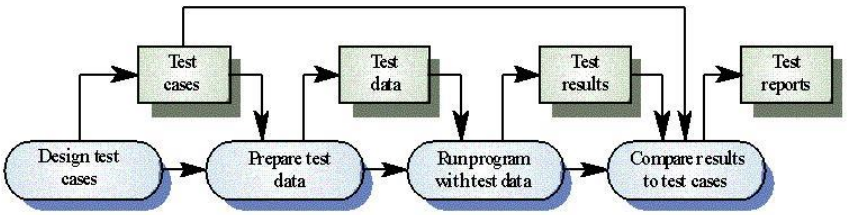
Software developer Independent testing team

The formality of testing

- Testing should be treated as a ***formal procedure:***
  - Inputs must be devised/prepared
  - Outcomes must be predicted
  - Test designs must be documented
  - Tests must be executed
  - Results must be observed and recorded
  - Results must be compared with predictions
- **The documentation is vital as part of a formal process**

Test data and test cases

- *Test data:* Inputs which have been devised to test the system
- *Predicted outputs/results:* The results that *should* be produced from the test data inputs if the system operates according to its specification
- *Test cases:* Record of the tests chosen to test the system, with rationale, data inputs and predicted outputs
- The formal testing process:



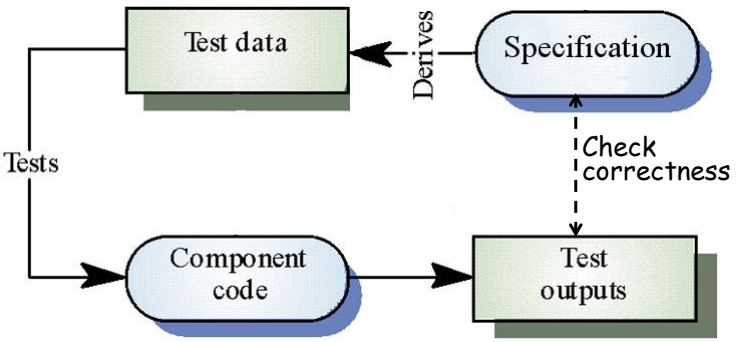
Diagrams from slides accompanying Somerville's text book

Test case design

- Test cases must be chosen/designed systematically
- Two main approaches to selecting test cases:
  - Black box testing
    - Knowing the *expected functions*...
    - ...design tests to check whether each function behaves as expected
  - White box testing (a.k.a. *glass box testing*)
    - Knowing the *internal operation of the code*...
    - ...design tests to exercise internal components, checking whether the overall behaviour is as expected

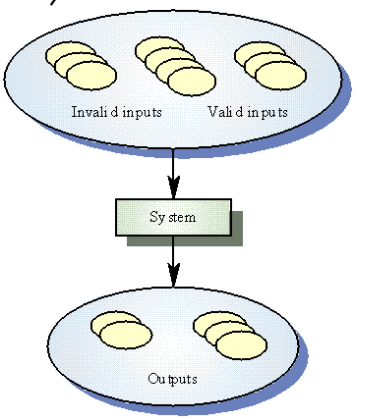
Black-box testing

- An approach to testing where the program, or a component, is considered as a 'black-box'
- The program test cases are based on the *system specification*
- Test planning can begin early in the software process



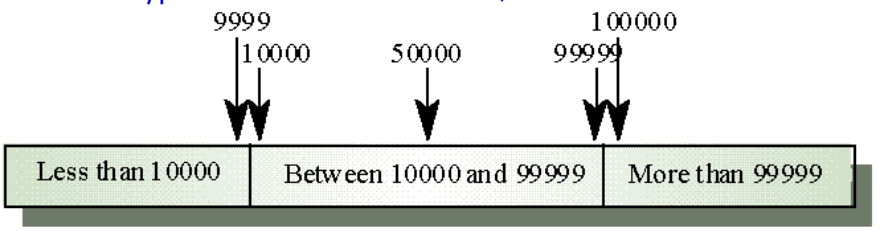
BB testing example: Equivalence partitioning

- Input data and output results often fall into *different classes* where all members of a class are related
- Each of these classes is an *equivalence partition* where the program behaves in an *equivalent way* for each class member
- Test cases should be chosen *from each partition*
  - It is also important to test input values at *boundaries* between partitions
  - And any known special values



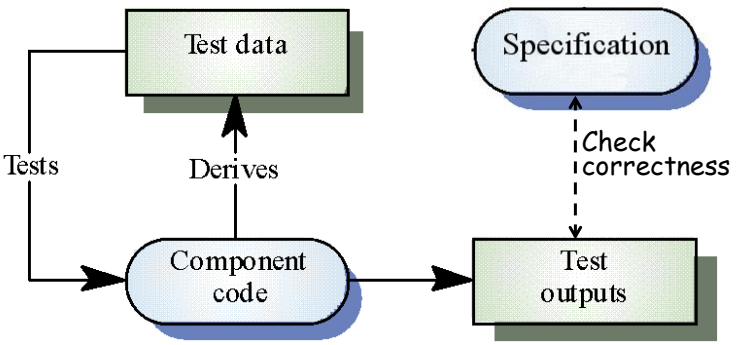
Equivalence partitioning: example

- Partition system inputs and outputs into 'equivalence sets':
  - Suppose the input is a 5-digit integer intended to be between 10000 and 99999
  - Then the equivalence partitions are
    - < 10000 (invalid)
    - 10000 - 99999 (typical)
    - > 99999 (invalid)
- We could choose test cases:
  - For typical values:
    - 50000, and maybe 11000, 90000
  - At the boundary of the partitions:
    - 9999, 10000, 99999, 10000
  - Typical invalid values: 5000, 150000



Structural testing

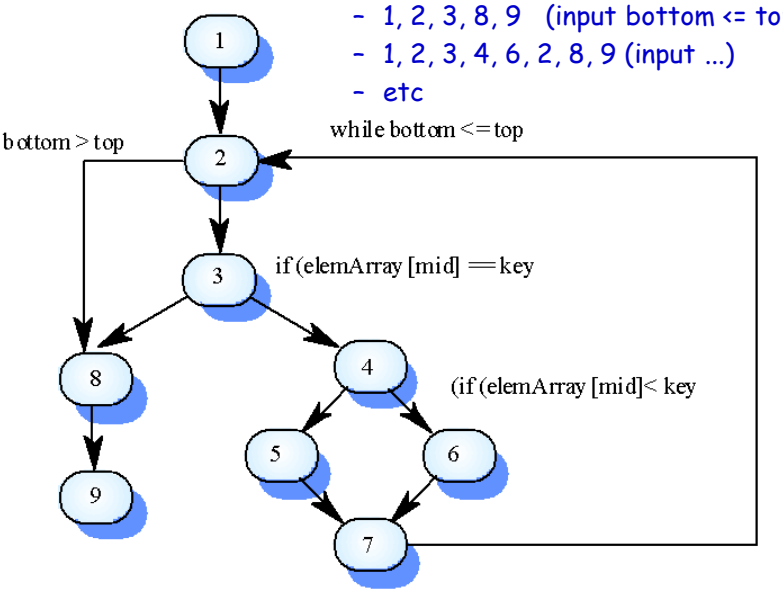
- Sometimes called *white-box* or *glass-box testing*
- Test cases are derived according to *program structure*
- The objective is to *exercise all program statements*
- Test data is selected to force execution of statements
  - Predicted outputs are determined for that test data
- The code is run with the test data – and actual outputs are compare with predicted outputs



WB testing example: Path testing

- The objective of path testing is to ensure that the set of test cases is such that each *path through the program* is executed at least once
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control
- Statements with conditions are therefore nodes in the flow graph

- Candidate paths:
  - 1, 2, 8, 9 (input bottom > top)
  - 1, 2, 3, 8, 9 (input bottom <= top and ...)
  - 1, 2, 3, 4, 6, 2, 8, 9 (input ...)
  - etc

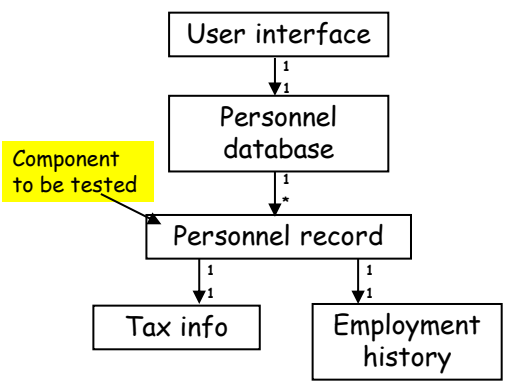


Unit testing

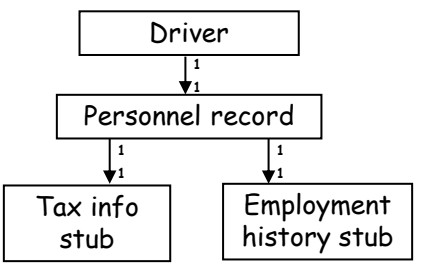
- Tests individual system components (classes, methods)
- Could be black-box or white box testing
- Since we are not testing a *complete system*, we must execute a *component* under test in a *test harness*
  - A software system configured to apply the test to just the specific component
  - May be a one-off for each test
- The test harness comprises "stubs" and/or "drivers"
  - Components that are *depended on* are replaced by *stubs*
  - A *stub* is an empty or dummy implementation of a component that respects its public interface
  - A *driver* organizes the tests – e.g. a JUnit test method
  - The driver instantiates the component under test and calls its methods – supplying test data, gathering and reporting the results

Example

Intended use of component:



Test harness:



End of lecture