# Software Testing: Overview

Conrad Hughes
School of Informatics

Slides thanks to Stuart Anderson

# Course Administration


- Main text, and others worth looking at:
  - Pezzè & Young, *Software Testing and Analysis: Process, Principles and Techniques*, Wiley, 2007.
  - G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1976, now in a second edition.
  - B. Marick, *The Craft of Software Testing*, Prentice Hall, 1995
  - C Kaner, J. Bach, B. Pettichord, *Lessons Learned in Software Testing*, Wiley, 2001
- Material covered via readings, presentations, web resources and practical experience.
- Conrad Hughes, Informatics Forum 3.46 [conrad.hughes@ed]
- Web page: http://www.inf.ed.ac.uk/teaching/courses/st/
- Useful: http://www.cs.uoregon.edu/~michal/book/index.html
- Useful: http://www.testingeducation.org

# Grading on the Course


- Two equal practicals worth 25% of the final mark. Practicals will involve actually testing some software systems.
- One examination worth 75%
- Quizzes and homeworks in the tutorials – not assessed but doing them will make it easier to do the examination and practicals.

## Famous person's quote time!

"…testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness."

- Edsger Dijkstra

[http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html]

# So really, why do we test?

- To find faults
  - Glenford Myers, *The Art of Software Testing*
- To provide confidence
  - of reliability
  - of (probable) correctness
  - of detection (therefore absence) of particular faults
- Other issues include:
  - Performance of systems (i.e. use of resources like time, space, bandwidth, …).
  - "…ilities" can be the subject of test e.g. usability, learnability, reliability, availability,
- Kaner and Bach: *a technical investigation carried out to expose quality-related information on the product under test.*

# Testing Theory


- But Dijkstra viewed programs as primarily abstract mathematical objects – for the tester they are engineered artifacts – the mathematics informs the engineering – but that is not the whole story (e.g. integers – a common trap for the unwary).
- Plenty of negative results
  - Nothing *guarantees* correctness
  - Statistical confidence is prohibitively expensive
  - Being systematic may not improve fault detection
    - as compared to simple random testing
  - Rates of fault detection don't correlate easily with measures of system reliability.
- Most problems to do with the "correctness" of programs are formally undecidable (e.g. program equivalence).

# What Information Do We Have Available?


- Specifications (formal or informal)
  - To check an output is correct for given inputs
  - for Selection, Generation, Adequacy of test sets
- Designs/Architecture
  - Useful source of abstractions
  - We can design for testability
  - Architectures often strive to separate concerns
- Code
  - for Selection, Generation, Adequacy
  - Code is not always available
  - Focus on fault/defect finding can waste effort
- Usage (historical or models) – e.g. in telecom traffic
- Organization experience – if the organisation gathers information

# Testing for Reliability

- Reliability is statistical, and requires a statistically valid sampling scheme
- Programs are complex human artifacts with few useful statistical properties
- In some cases the environment (usage) of the program has useful statistical properties
  - Usage profiles can be obtained for relatively stable, pre-existing systems (telephones), or systems with thoroughly modeled environments (avionics)

# A Hard Case: Certifying Ultra-High Reliability


- Some systems are required to demonstrate very high reliability (e.g. an aircraft should only fail completely once in $10^{11}$ hours of flying).
- So aircraft components have to be pretty reliable (but think about how many single points of failure a car has).
- How can we show that the avionics in a fly-by-wire aircraft will only fail once in $10^{9}$ hours of flying (so there is a way to fly without avionics).
- Butler & Finelli estimate
  - for $10^{-9}$ per 10 hour mission
  - requires: $10^{10}$ hours testing with 1 computer
  - or: $10^{6}$ hours (114 years) testing with 10,000 computers

[ACM Sigsoft 91, Conf. on SW for Critical Systems]

[also Littlewood and Strigini, *Validation of ultrahigh dependability for software-based systems*, CACM, 69-80, vol 36, no 11, 1993.]

# Standard Testing Activities

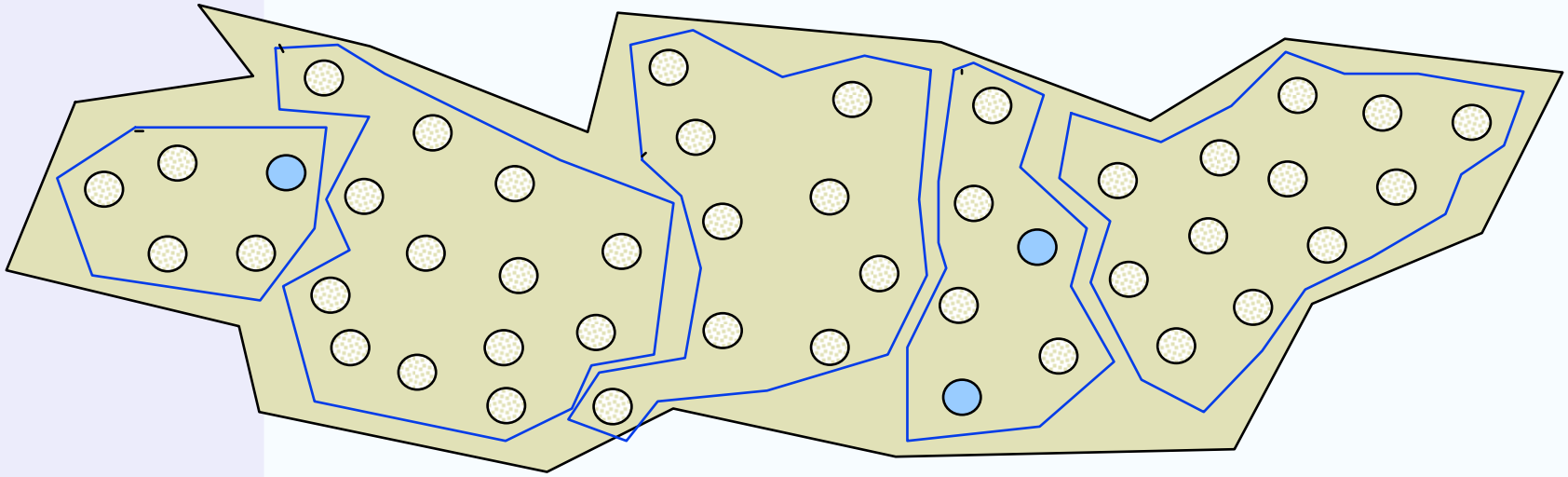

- Phase 1: Modelling the environment of the software
  - What is the right abstractions for the interface?
- Phase 2: Selecting test scenarios
  - How shall we select test cases?
    - Selection; generation
- Phase 3: Running and evaluating test scenarios
  - Did this test execution succeed or fail?
    - Oracles
  - What do we know when we're finished?
    - Assessment
- Phase 4: Measuring testing progress
  - How do we know when we've tested enough?
    - Adequacy

# Phase 1: Modelling the Environment


- Testers identify and simulate interfaces that a software system uses
- Common interfaces include:
  - Human interfaces
  - Software interfaces (aka APIs)
  - File system interfaces
  - Communication interfaces
- Identify interactions that are beyond the control of the system, e.g.
  - Hardware being powered off and on unexpectedly
  - Files being corrupted by other systems/users
  - Contention between users/systems
- Issues in building abstractions include: choosing representative values, combinations of inputs, sequence (finite state machine models are often used)

# Phase1: Partition the Input Space

- Basic idea: Divide program input space into (what we think might be) equivalence classes
  - Use representatives of the "equivalence classes" to model the domain
  - Worry about the boundaries because we don't know if we have the right partition.



# Phase 1: Specification-Based Partition Testing


- Divide the program input space according to cases in the specification
  - May emphasize boundary cases
  - Combining domains can create a very large number of potential cases.
  - Abstractions can lose dependencies between inputs
- Testing could be based on systematically "covering" the categories
  - The space is very large and we probably still need to select a subset.
  - May be driven by scripting tools or input generators
  - Example: Category-Partition testing [Ostrand]
- Many systems don't have particularly good specifications.
- Some development approaches use tests as a means of specification.

# Quiz: Testing Triangles (G. Myers)


- You are asked to test a method Triangle.scalene(int,int,int) that returns a boolean value.
- Triangle.scalene(p,q,r) is true when p, q and r are the lengths of the sides of a *scalene* triangle.
- Scalene as opposed to equilateral or isosceles
- Construct an adequate test set for such a method.

# Quiz: Rate Yourself


1. A *valid* scalene triangle (e.g. 4,3,2)
2. A *valid* equilateral triangle.
3. A *valid* isosceles triangle (e.g. 2,4,4 *not 4,2,2*)
4. Permuted isosceles inputs (e.g. 2,4,4; 4,2,4; 4,4,2)
5. Zero side length?
6. Negative side lengths?
7. Inputs such that p=q+r
8. Permutations of test cases 7.
9. Inputs such that p > q+r
10. Permutations of test cases 9.
11. All zero?
12. Did you specify the expected result in all cases?
13. If we had an interface to the function there would be many more.

# Quiz: Does having the code help? [1]

```
public class Triangle {
    public boolean scalene(int p, int q, int r) {
      int tmp;
      if (q>p){tmp = p; p = q; q = tmp;}
      if (r>p){tmp = p; p = r; r = tmp;}
        return((r>0)&&(q>0)&&(p>0)&&
               (p<(q+r))&& ((q>r)||(r>q)));
    }
}
```

# Quiz: Does having the code help? [2]

```
public class Triangle {
    public boolean scalene(int p, int q, int r) {
      if(q > p) SWAP(p, q);
      if(r > p) SWAP(p, r);
      if(r > q) SWAP(q, r);
      return (r > 0) && (p < q + r) &&
          (q < r) && (r < p);
    }
}
```

## Quiz: Summary


- The code is less than 10 lines long – we seem to need at least the same number of tests to check it.
- Many modern systems are multi-million line systems.
- Daunting task to work out how to test such systems.
- Part of the approach is to change the way systems are built.

# Doomed software project time!

"Vice President Jim Allchin, personally broke the bad news to Bill Gates. Allchin is co-head of the Platform Products and Services Division. "It's not going to work," he told Gates in the chairman's office mid-2004, the paper reports. "[Longhorn] is so complex its writers will never be able to make it run properly. "The reason: Microsoft engineers were building it just as they had always built software. Thousands of programmers each produced their own piece of computer code, to be stitched together into one sprawling program. But Longhorn/Vista was too complex: Microsoft needed to begin again, Allchin told Gates. "

# Phase 2: Selecting Tests


- What criteria can we use to cut down the number of tests.
- Common criteria are *coverage criteria:*
  - We have executed all statements.
  - We have executed all branches
  - We have executed all possible paths in the program
  - We have covered all possible data flows.
- We might also try to evaluate the effectiveness of test cases by seeding errors in the code and seeing how well a test set does in finding the errors.
- We might also consider statistical measures e.g. that we have a statistically valid sample of the possible inputs (but here we need a good idea of the distribution of inputs).

# Phase 2: Test Adequacy


- Ideally: adequate testing ensures some property (proof by cases)
  - Origins in [Goodenough & Gerhart], [Weyuker and Ostrand]
  - It is very hard to establish non-trivial properties using these methods (unless the system is clearly finite)
- Practical "adequacy" criteria are safety measures designed to identify holes in the test set:
  - If we have not done this kind of test some instances of this kind of test should be added to the test set.

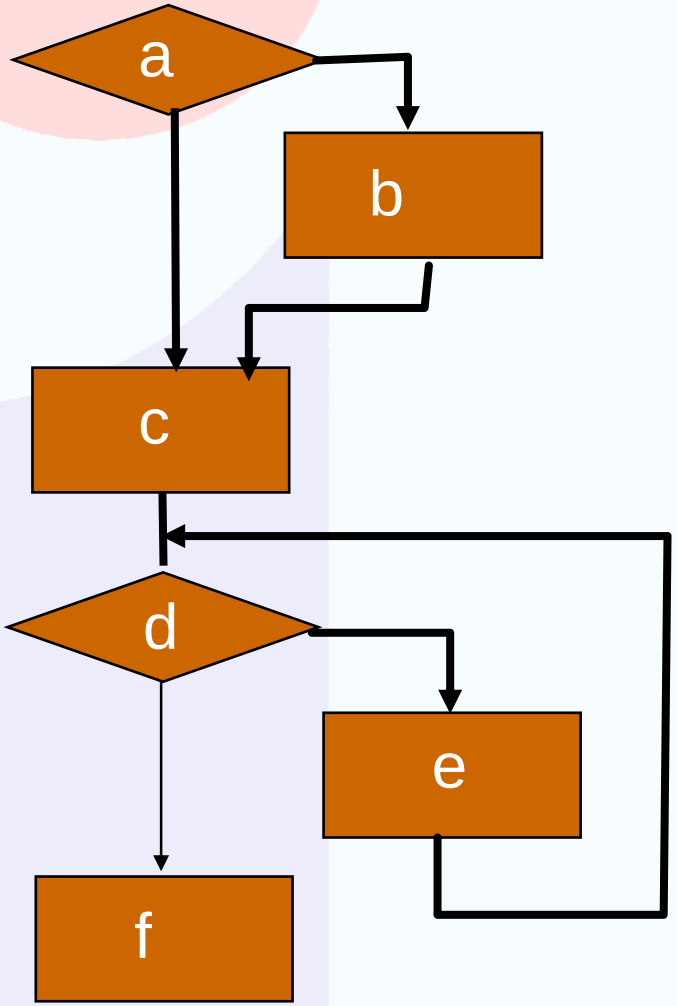
# Phase 2: Systematic Testing


- Systematic (non-random) testing is aimed at program improvement, i.e. finding faults *not* trying to predict the statistical behaviour of the program
  - Obtaining valid samples and maximizing fault detection require different approaches; it is unlikely that one kind of testing will be satisfactory for both
- "Adequacy" criteria mostly negative: indications of important omissions
  - Positive criteria (assurance) are no easier than program proofs

# Phase 2: Structural Coverage Testing


- (In)adequacy criteria
  - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) and condition coverage
  - Data flow (syntactic dependency) coverage
  - Various control-flow criteria
- Attempted compromise between the impossible and the inadequate

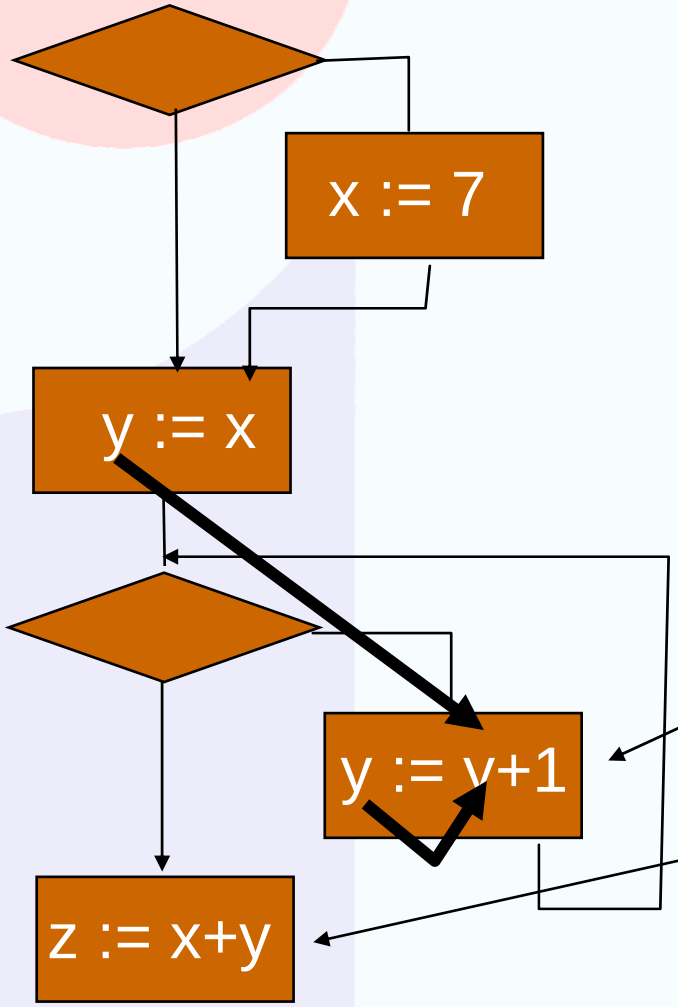
# Phase 2: Basic structural criteria




Edge ac is required by all-edges but not by all-nodes coverage

Typical loop coverage criterion would require zero iterations (cdf), one iteration (cdedf), and multiple iterations (cdededed...df)

# Phase 2: Data flow coverage criteria




*Rationale: An untested def-use association could hide an erroneous computation*

2 reaching definitions (one is from self)

2 reaching definitions for x, and 2 reaching definitions for y

# Phase 2: Structural Coverage in Practice


- Statement and sometimes edge or condition coverage is used in practice
  - Simple lower bounds on adequate testing; may even be harmful if inappropriately used for test selection – too much focus on structure diverts effort from bugs that worry users
- Additional control flow heuristics sometimes used
  - Loops (never, once, many), combinations of conditions
  - Potential linkage to static flow analysis literature
- Slicing and abstract interpretation approaches allow the checking of basic properties on large bodies of code (e.g. Airbus 380 avionics ~3-4 Mloc)

# Phase 2: Fault-based testing

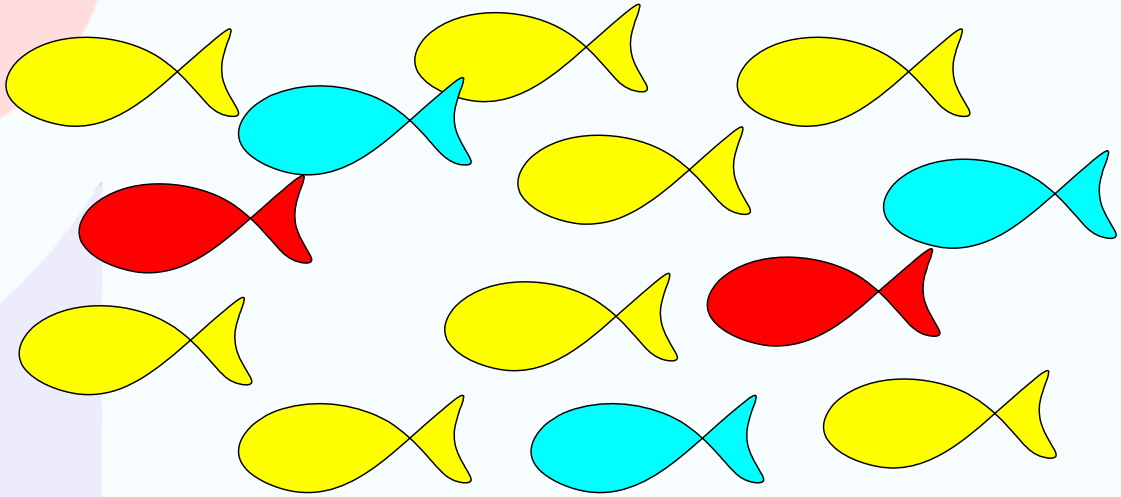

- Given a fault model
  - hypothesized set of deviations from correct program
  - typically, simple syntactic *mutations*; relies on coupling of simple faults with complex faults
- Coverage criterion: Test set should be adequate to reveal (all, or x%) faults generated by the model
  - similar to hardware test coverage

# Phase 2: Fault Models


- Fault models are key to semiconductor testing
  - Test vectors graded by coverage of accepted model of faults (e.g., "stuck-at" faults)
- What are fault models for software?
  - What would a fault model look like?
  - How general would it be?
    - Across application domains?
    - Across organizations?
    - Across time?
- Defect tracking is a start – gathering collections of common faults in an organisation – rigorous process – links to Capability Maturity Model and optimising organisations.

# Phase 2: Selection vs. Adequacy

## Mutation Testing Example



- Red fish = real program faults (unknown population)
- Blue fish = seeded faults (e.g., mutations) or representative behaviors (known population)
- Adequacy: count blue fish caught, estimate red fish
- Misuse for selection: use special bait to catch blue fish

# Phase 2: Test Selection: Standard Advice


- Specification coverage is good for selection as well as adequacy
  - applicable to informal as well as formal specs
- Fault-based tests
  - usually ad hoc, sometimes from check-lists
- Program coverage last
  - to suggest uncovered cases, not just to achieve a coverage criterion

## Phase 2: The Bottom Line: The Budget Coverage Criterion


- A common answer to "when is testing done"
  - When the money is used up
  - When the deadline is reached
- This is sometimes a rational approach!
  - Implication 1: Test selection is more important than stopping criteria per se.
  - Implication 2: Practical comparision of approaches must consider the cost of test case selection
- Example: testing of SAFEBUS – started out with a pile of money and stopped when they ran out (could have more money if it was still flakey).

# Phase 3: Running and Evaluating Tests


- The magnitude of the task is a problem than can require tools to help – automated testing means we can do more testing but in some circumstances it is hard (e.g. GUIs)
- Is the answer right? Usually called the Oracle problem – often the oracle is human.
- Two approaches to improving evaluation: better specification to help structure testing; embedded code to evaluate structural aspects of testing (e.g. providing additional interfaces to normally hidden structure.
- Through life testing: most programs change (some are required not to change by law) – regression testing is a way of ensuring the next version is a least as good as the previous one.
- Reproducing errors is difficult – attempt to record sequence of events and replay – issues about replicating the environment.

# Phase 3: The Importance of Oracles


- Much testing research has concentrated on adequacy, and ignored oracles
- Much testing practice has relied on the "eyeball oracle"
  - Expensive, especially for regression testing
    - makes large numbers of tests infeasible
  - Not dependable
- Automated oracles are essential to cost-effective testing

# Phase 3: Sources of Oracles


- Specifications
  - sufficiently formal (e.g., SCR tables)
  - but possibly incomplete (e.g., assertions in Anna, ADL, APP, Nana)
- Design, models
  - treated as specifications, as in protocol conformance testing
- Prior runs (capture/replay)
  - especially important for regression testing and GUIs; hard problem is parameterization

# Phase 3: What can be automated?

- Oracles
  - assertions; replay; from some specifications
- Selection (Generation)
  - scripting; specification-driven; replay variations
  - selective regression test
- Coverage
  - statement, branch, dependence
- Management

# Phase 3: Design for Test: 3 Principles

Adapted from circuit and chip design

- Observability
  - Providing the right interfaces to observe the behavior of an individual unit or subsystem
- Controllability
  - Providing interfaces to force behaviors of interest
- Partitioning
  - Separating control and observation of one component from details of others

# Phase 4: Measuring Progress (Are we done yet?)


- Structural:
  - Have I tested for common programming errors?
  - Have I exercised all of the source code?
  - Have I forced all the internal data to be initialized and used?
  - Have I found all seeded errors?
- Functional:
  - Have I thought through the ways in which the software can fail and selected tests that show it doesn't?
  - Have I applied all the inputs?
  - Have I completely explored the state space of the software?
  - Have I run all the scenarios that I expect a user to execute?

# Summary

- We have outlined the main activities in testing activity:
  - Modelling the environment
  - Test Selection
  - Test execution and assessment
  - Measuring progress
- These are features of all testing activity.
- Different application areas require different approaches
- Different development processes might reorganise the way we put effort into test but the amount of test remains fairly constant for a required level of product quality.

# Acknowledgements


- Michal Young's overview of software testing.
- James A. Whittaker's What is Software Testing…
- Brad Meyer's Art of Software testing for the scalene triangle example