

---

# Software Testing: Overview

Stuart Anderson



# Title Slide

## **Acknowledgements:**

Massimo Felici, Conrad Hughes

## Course Administration: Books

- **Main text: Pezzè & Young, Software Testing and Analysis: Process, Principles and Techniques, Wiley, 2007.**
- Paul Ammann and Jeff Offutt, Introduction to Software Testing, Cambridge University Press, Cambridge, UK, ISBN 0-52188-038-1, 2008.
- G.J. Myers, The Art of Software Testing, Second Edition, John Wiley & Sons, New York, 1976.
- 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.

## Course Administration

- **Course 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

- **One practicals worth 25% of the final mark** — Practicals will involve actually testing some software systems
- **Deadline:** Thursday, 1600, 1 March 2012 (week 7)
- In week 8 each group will organise a 30 minute feedback session to demonstrate their practical and get feedback.
- **One examination worth 75%.** This will be an **open-book** examination.
- **Quizzes and tutorials** — not assessed but doing them will make it much easier to do the examination and practicals

## Tutorials

- There are four tutorials available on the course. Each one is *owned* by a different tutor.
- Each tutorial relates to a different section of the practical.
- To access the tutorial you must have evidence of preparation for the tutorial (e.g. your group has completed a small test or has some documentation available).
- When you are ready to do a tutorial you contact the tutor to arrange a time for a tutorial session.
- Each tutorial session will have two groups participating.

## Famous persons 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
- **Organisation 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 modelled 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.
- 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  
[also Littlewood and Strigini]

# Slide 10: A Hard Case: Certifying Ultra-High Reliability

## Suggested Readings

- Butler, R.W.; Finelli, G.B., The infeasibility of quantifying the reliability of life-critical real-time software, IEEE Transactions on Software Engineering, vol.19, no.1, pp.3-12, Jan 1993.

DOI: <http://dx.doi.org/10.1109/32.210303>

- Bev Littlewood and Lorenzo Strigini. 1993. Validation of ultrahigh dependability for software-based systems. Commun. ACM 36, 11 (November 1993), 69-80.

DOI: <http://dx.doi.org/10.1145/163359.163373>

# Standard Testing Activities

**Phase 1:** Modelling the environment of the software

- What is the right abstraction 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 have finished?
  - Assessment

**Phase 4:** Measuring testing progress

- How do we know when we have tested enough?
  - Adequacy

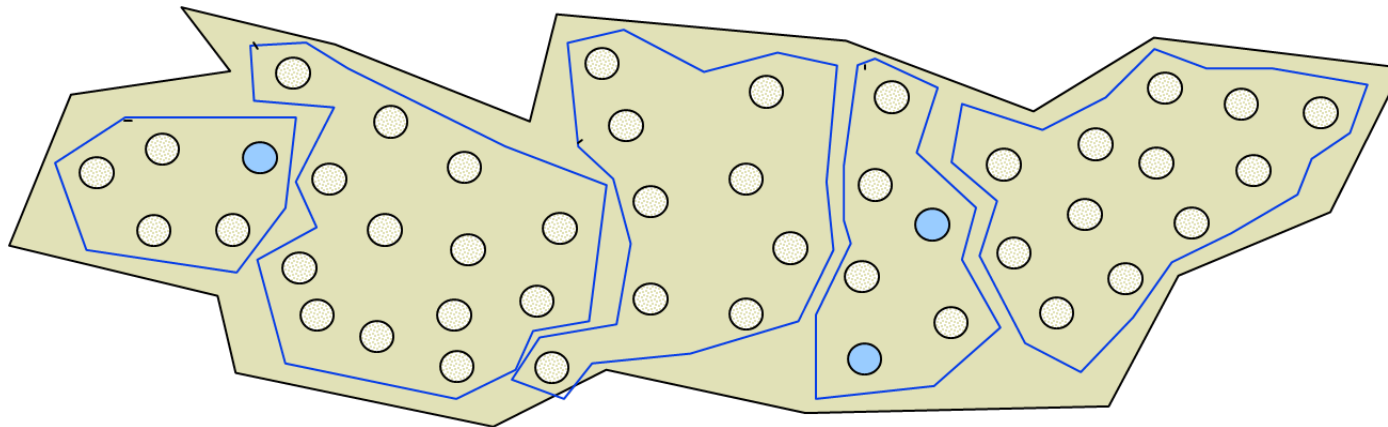
## Phase 1: Modelling the Environment

- *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)

## Phase 1: 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 do not 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 emphasise 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 do not 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.

## Slide 15: 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?

```
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)));
    }
}
```

**Note: this code contains at least one bug!**

## 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!

*“Jim Allchin walked into Bill Gates office to deliver a bombshell about the next generation of Microsoft Windows. ”Its not going to work,” he told the Microsoft chairman. The new version, code-named Longhorn, was so complex its writers would never be able to make it run properly. Worse: Longhorn was irredeemable because Microsoft engineers were building it just as they had always built software. Throughout its history, Microsoft had let thousands of programmers each produce their own piece of computer code, then stitched it together into one sprawling program. Now, Mr. Allchin argued, the jig was up. Microsoft needed to start over.”*

– Wall Street Journal, 23 September 2005

## 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)
  - It is very hard to establish non-trivial properties using these methods (unless the system is clearly finite)
  - Origins in [Goodenough and Gerhart], [Weyuker and Ostrand]
- **Practically:** “*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.



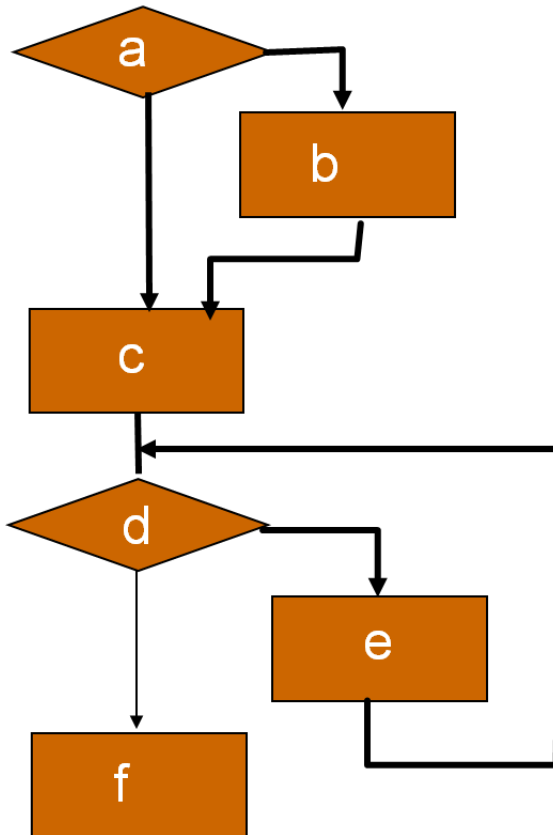
## Phase 2: Systematic Testing

- Systematic (non-random) testing is aimed at program improvement
  - Finding faults not trying to predict the statistical behaviour of the program
  - Obtaining valid samples and maximising 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

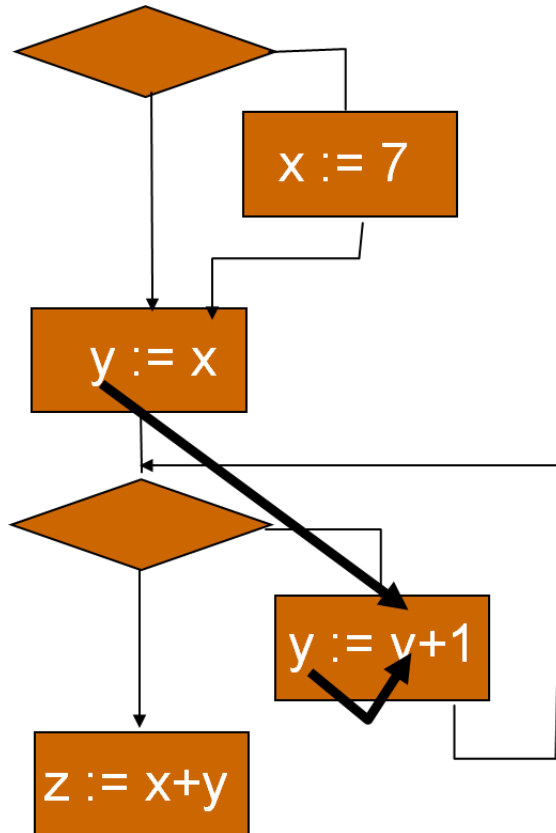
- (In)adequacy criteria
  - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) and condition coverage
  - Data flow (syntactic dependency) coverage
  - Various control-flow 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 A380 avionics  $\sim 3\text{-}4$  Mloc; modern luxury car  $\sim 100$ Mloc)

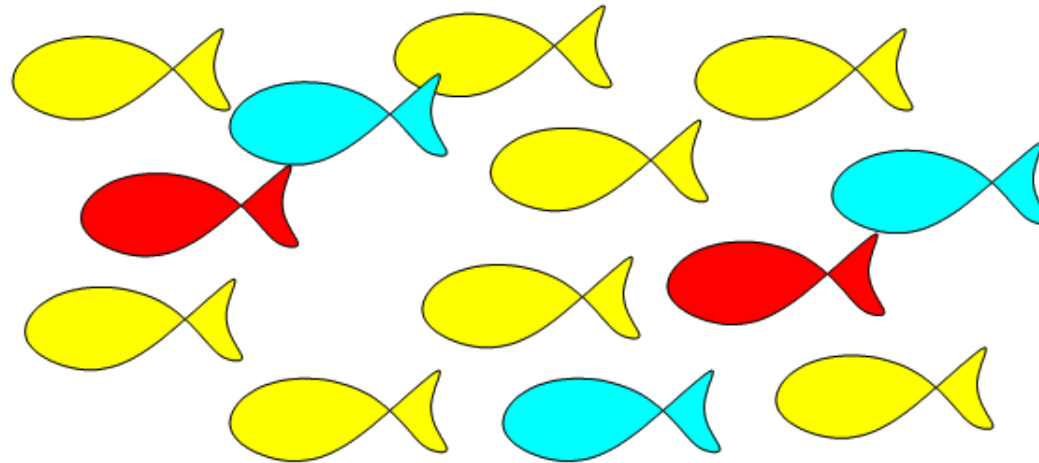
## Phase 2: Fault-Based Testing

- Given a fault model
  - hypothesised 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 organisations? Across time?
- Defect tracking is a start — gathering collections of common faults in an organisation — rigorous process — links to CMMI (Capability Maturity Model Integration) and optimising organisations.

## Phase 2: Selection vs. Adequacy



- **Red fish** = real program faults (unknown population)
- **Blue fish** = seeded faults (e.g., mutations) or representative behaviour (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 Advise

- **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 finished?*’
  - 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 comparison of approaches must consider the cost of test case selection
- Example: testing of SAFEBUS (communications bus for Boeing aircraft) — 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 unfeasible
  - 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 embedded assertion languages such as 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 — Three Principles

### 1. Observability

- Providing the right interfaces to observe the behavior of an individual unit or subsystem

### 2. Controllability

- Providing interfaces to force behaviours of interest

### 3. 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 initialised 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 testing activities:
  - 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.

# Readings

## Required Readings

- **Textbook (Pezzè and Young):** Chapter 1, Software Test and Analysis in a Nutshell
- **Textbook (Pezzè and Young):** Chapter 2, A Framework for Test and Analysis
- Whittaker, J.A., What is software testing? And why is it so hard?, IEEE Software, vol.17, no.1, pp.70-79, Jan/Feb 2000.

DOI: <http://dx.doi.org/10.1109/52.819971>

## Slide 38: Acknowledgements

- Michal Youngs overview of software testing.
- James A. Whittakers What is Software Testing...
- Brad Meyers Art of Software testing for the scalene triangle example