Software Engineering 4

# The Software Testing Life-Cycle


Andrew Ireland

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh




## Why Test?

- Devil's Advocate:

  *"Program testing can be used to show the presence of defects, but never their absence!"*

  Dijkstra

  *"We can never be certain that a testing system is correct."*

  Manna

- In Defence of Testing:
  - Testing is the process of showing the presence of defects.
  - There is no absolute notion of "correctness".
  - Testing remains the most cost effective approach to building confidence within most software systems.

## Executive Summary

*A major theme of this module is the integration of testing and analysis techniques within the software life-cycle. Particular emphasis will be placed on code level analysis and safety critical applications. The application and utility of static checking will be studied through extensive use of a static analysis tool (ESC Java) for Java.*



## Low-Level Details

- Lecturers: Lilia Georgieva (G54) and Andrew Ireland (G57) [ `lilia@macs.hw.ac.uk` and `a.ireland@hw.ac.uk` ]
- Class times:
  - Tuesday 3.15pm EC 3.36
  - Thursday 3.15pm EC 2.44
  - Friday 10.15 EC 2.44 (Lecture/Workshop) EC 2.50 (Lab)
  - Friday 11.15 EC 2.50 (Lab)

  **Format of Friday classes will vary from week-to-week.**
- Web: `http://www.macs.hw.ac.uk/~air/se4/`
- Assessment:
  - Separate assignments for CS and IT streams.
  - Overall assessment: exam (75%) coursework (25%).

## Software Testing and Analysis Thread

- **The Software Testing Life-Cycle:**
  A broad introduction to the role of testing within software development – practical exercises in requirements testing.
- **Dynamic Analysis:**
  A review of dynamic analysis techniques as used for code level verification – practical exercises in dynamic analysis.
- **Static Analysis:**
  A review of static analysis techniques within the software development life cycle – practical exercises in static analysis.
- **Safety Critical Systems:**
  An introduction to the software issues that arise when developing systems where failure can lead to loss of life – case study material from real-world applications will be reviewed.



## A Historical Perspective

- In the early days (1950's) you wrote a program then you tested and debugged it. Testing was seen as a follow on activity which involved detection and correction of coding errors, *i.e.*

  Design ⇒ Build ⇒ Test

  Towards the late 1950's testing began to be decoupled from debugging — but still seen as a post-hoc activity.
- In the 1960's the importance of testing increased through experience and economic motivates, *i.e.* the cost of recovering from software deficiencies began to play a significant role in the overall cost of software development. More rigorous testing methods were introduced and more resources made available.

## A Historical Perspective

- In the 1970's "software engineering" was coined. Formal conferences on "software testing" emerged. Testing seen more as a means of obtaining confidence that a program actually performs as it was intended.
- In the 1980's "quality" became the big issue, as reflected in the creation of the IEEE, ANSI and ISO standards.
- In the 1990's the use of tools and techniques more prevalent across the software development life-cycle.



## But What is Software Testing?

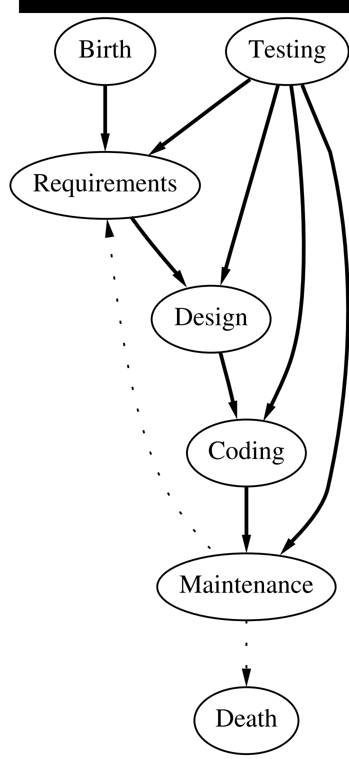
- *"Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements, or to identify differences between expected and actual results."* IEEE
- *"The process of executing a program or system with the intent of finding errors."* (Myers 1979)
- *"The measurement of software quality."* (Hetzel 1983)

## What Does Testing Involve?

- Testing = Verification + Validation
- Verification: building the product right.
- Validation: building the right product.
- A broad and continuous activity throughout the software life cycle.
- An information gathering activity to enable the evaluation of our work, *e.g.*
  - Does it meet the users requirements?
  - What are the limitations?
  - What are the risks of releasing it?



## Testing is for "Life"



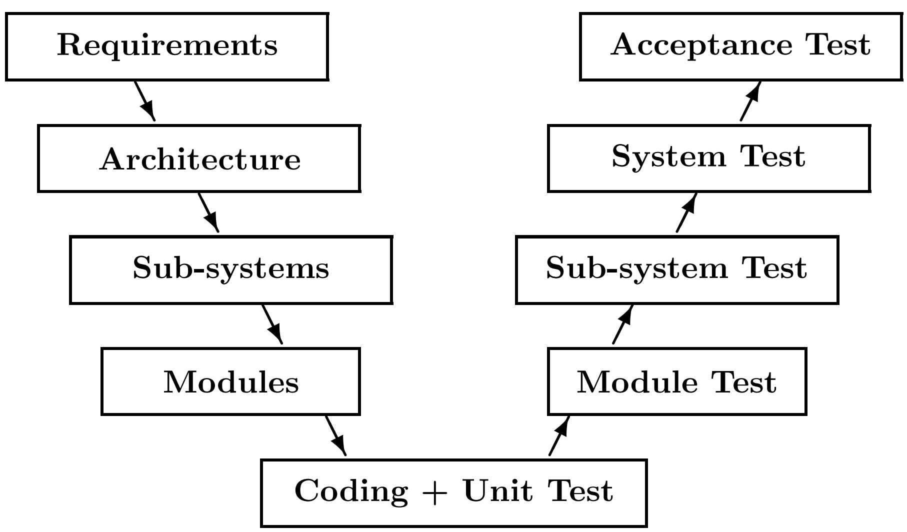

Early identification of defects & prevention of defect migration are key goals of the testing process.

## Some Key Issues

- A time limited activity:
  - Exhaustive testing not possible.
  - Full formal verification not practical.
- Must use the time available intelligently.
- Must clearly define when the process should stop.
- Ease of testing versus efficiency:
  - Programming language issues.
  - Software architectural issues.
- Explicit planning is essential!



## V Software Life-cycle Model



## Requirements Testing

**Unambiguous:** Are the definitions and descriptions of the required capabilities precise? Is there clear delineation between the system and its environment?

**Consistent:** Freedom from internal & external contradictions?

**Complete:** Are there any gaps or omissions?

**Implementable:** Can the requirements be realized in practice?

**Testable:** Can the requirements be tested effectively?



## Requirements Testing

- 80% of defects can be typically attributed to requirements.
- Late life-cycle fixes are generally costly, *i.e.* 100 times more expensive than corrections in the early phases.
- Standard approaches to requirements testing & analysis:
  - "Walk-throughs" or Fagan-style inspections (more detail in the static analysis lecture).
  - Graphical aids, *e.g.* cause-effect graphs, data-flow diagrams.
  - Modelling tools, *e.g.* simulation, temporal reasoning.

  Note: modelling will provide the foundation for high-level design.

## Planning for Testing

- Forward planning crucial for estimating and minimizing costs.
- The plan should identify:
  - which aspects of the system should be tested.
  - a criteria for success.
  - the methods and techniques to be used.
  - personnel responsible for the testing.
- Mechanisms for recording, tracking and analyzing defects are crucial to project planning and management.

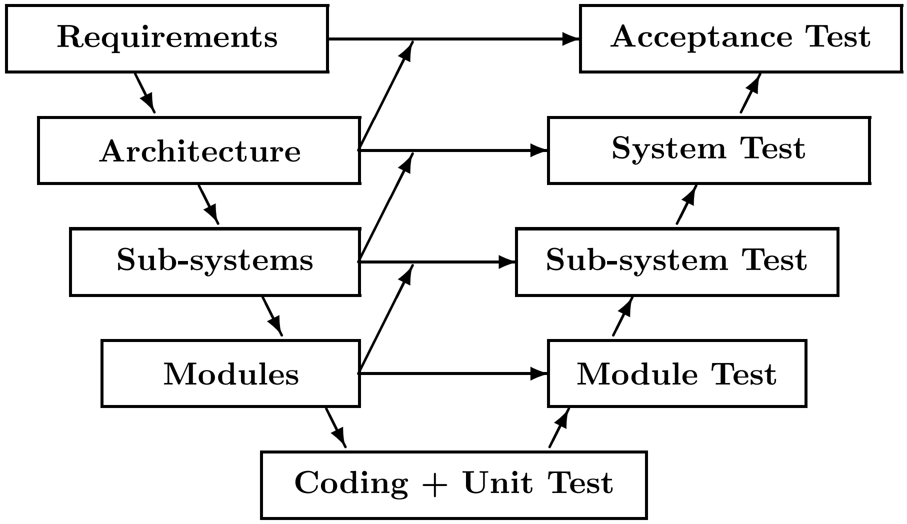


## Requirements Trace-ability

| **Requirement** | **Sub-system** | **Module** | **Code** | **Tests** |
|---|---|---|---|---|
| reverse-thruster activation conditional on landing gear deployment | Avionics controller | `EngineCtrl` | Lines 100,239 | 99,101 |
| | | `BrakeCtrl` | Lines 52,123 | 11,51 |
| ... | ... | ... | ... | ... |

Volatility of requirements calls for systematic tracking through to code level test cases.

## Planning for Testing






## Design Testing

- Getting the system architecture right is often crucial to the success of a project. Alternatives should be explored explicitly, *i.e.* by review early on in the design phase.
- Without early design reviews there is a high risk that the development team will quickly become locked into one particular approach and be blinkered from "better" designs.
- Where possible, executable models should be developed in order to evaluate key design decisions, *e.g.* communication protocols. Executable models can also provide early feedback from the customer, *e.g.* interface prototypes.
- Design-for-test, *i.e.* put in the "hooks" or "test-points" that will ease the process of testing in the future.

## Exploiting Design Notations: UML

**Object Constraint Language (OCL):** provides a language for expressing conditions that implementations must satisfy (feeds directly into unit testing – dynamic analysis lecture).

**Use Case Diagrams:** provides a user perspective of a system:

- Functionality
- Allocation of functionality
- User interfaces

Provides a handle on defining equivalence classes for unit testing (dynamic analysis lecture).



## Exploiting Design Notations: UML

**State Diagrams:** provides a diagrammatic presentation for a finite state representation of a system. State transitions provide strong guidance in testing the control component of a system.

**Activity Diagrams:** provides a diagrammatic presentation of activity co-ordination constraints within a system. Synchronization bars provide strong guidance in testing for key co-ordination properties, *e.g.* the system is free from dead-lock.

**Sequence Diagrams:** provides a diagrammatic presentation of the temporal ordering of object messages. Can be used to guide the testing of both synchronous and asynchronous systems.

## Code & Module Testing

Unit testing is concerned with the low-level structure of program code. The key objectives of module and unit testing are:

- Does the logic work properly?
  - Does the code do what is intended?
  - Can the program fail?
- Is all the necessary logic present?
  - Are any functions missing?
  - Is there any "dead" code?

Note: Code and module testing techniques will be the focus of static and dynamic analysis lectures.



## Sub-System Testing

- Focuses on the integration and testing of groups of modules which define sub-systems – often referred to as integration testing.
- Non-incremental or "big bang" approach:
  - Costly on environment simulation, *i.e.* stub and driver modules.
  - Debugging is non-trivial.
- Incremental approach:
  - Fewer stub and driver modules.
  - Debugging is more focused.
- Strategies: top-down, bottom-up, function-based, thread-based, critical-first, opportunistic.

## Testing Interfaces

**Interface misuse:** type mismatch, incorrect ordering, missing parameters – should be identified via basic static analysis.

**Interface misunderstanding:** the calling component or client makes incorrect assumptions about the called component or server – can be difficult to detect if behaviour is mode or state dependent.

**Temporal errors:** mutual exclusion violations, deadlock, liveness issues – typically very difficult to detect, model checking provides one approach.



## System Testing

**Volume and stress testing:** Can the system handle the required data throughput, requests etc? What are the upper bounds?

**Configuration testing:** Does the system operate correctly on all the required software and hardware configurations?

**Resource management testing:** Can the system exceed memory allocation limits?

**Security testing:** Is the system secure enough?

**Recovery testing:** Use pathological test cases to test system recovery capabilities.

**Availability/reliability:** Does the system meet the requirements?

## Acceptance Testing

- The objective here is to determine whether or not the system is ready for operational use.
- Focuses on user requirements and user involvement is high since they are typically the only people with "authentic" knowledge of the systems intended use.
- Test cases are typically designed to show that the system does **not** meet the customers requirements, if unsuccessful then the system is accepted.
- Acceptance testing is very much to do with *validation*, *i.e.* have we built the right product, rather than *verification*, *i.e.* have we built the product right.



## Change Management & Testing

- Reasons for change:
  - Elimination of existing defects.
  - Adaptation to different application environments,
  - Alteration in order to improve the quality of the product.
  - Extensions in order to meet new requirements.
- Testing for change:
  - Determine if changes have regressed other parts of the software – regression testing.
  - Cost-risk analysis: full regression testing or partial regression testing?
  - Effectiveness: automation and persistent test-points.

## Summary

- The testing life-cycle.
- Prevention better than cure – testing should start early both in terms of immediate testing and planning for future testing.
- Planning is crucial given the time-limited nature of the testing activity – planning should be, as far as possible, integrated within your design notations and formalisms.



## References

- "The Art of Software Testing", Myers, G.J. Wiley & Sons, 1979.
- "The Complete Guide to Testing", Hetzel, B. QED Information Sciences Inc, 1988.
- "Software Testing in the Real World", Kit, E. Addison-Wesley, 1995.
- "The Object Constraint Language: precise modeling with UML", Warmer, J. & Kleppe, A. Addison-Wesley, 1998.
- IEEE Standard for Software Test Documentation, 1991 (IEEE/ANSI Std 829-1983)
- IEEE Standard for Software Verification and Validation Plans, 1992 (IEEE/ANSI Std 1012-1986)