Software Design (F28SD2)
Verification & Validation

Andrew Ireland
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Verification & Validation

- Verification:
  - building the product right.
  - ensuring the software implements the design.
- Validation:
  - building the right product.
  - ensuring the software achieves the customer's requirements.
- A broad set of activities that span the software life cycle.
- An evidence 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?

Focus on Testing: 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 is not limited to code level execution ...

Static and Dynamic Perspectives

- Static Perspective:
  - techniques which do not involve executing code.
  - provide early feedback on design and implementation decisions.
- Dynamic Perspective:
  - techniques which involve executing code.
  - involves executing the code on specific data and analyzing the results – typically referred to as **testing**.

Note: both approaches have strengthens and weaknesses, so should be seen as complementary. But dynamic testing remains the predominate approach.

Testing and Debugging

- Debugging:
  - A developer led process that aims to identify and correct bugs. It may also involve some level of checking to ensure bug fixes have been correctly implemented.
- Testing:
  - A systematic analysis of a component or system with the aim of identifying bugs as well as ensuring conformance to requirements and/or specifications.
  - A systematic analysis to ensure that bug fixes have been implemented correctly, and have not regressed the system as a whole, *i.e.* regression testing.

Testing is for "Life"

- The causes of many software bugs, and certainly the most expensive ones, can be traced to requirements and/or design decisions.
- Early identification of bugs and prevention of bug migration are therefore key goals of a test process.
- Testing and analysis should therefore span the software life cycle, i.e.
  - Requirements
  - Design
  - Coding
  - Maintenance
- A systematic approach is required ...

The Process of Testing

Planning and control:

- Exhaustive testing is not possible, so the available time needs to be used effectively – planning is required in order to identify:
  - and prioritize what needs to be tested
  - what level of testing (evidence) will be required
  - an exit or success criteria
  - personnel responsible for each testing activity
- Control is required when reality, i.e. what is actually achieved, deviates from the plan.
- Effective control requires a mechanism for recording, tracking and analyzing defects.

The Process of Testing

Analysis and design:

- Focus here is on deciding what tests are required, the aim being to have as few test cases as possible – test cases need to be maintained, so the fewer high yield cases you require the cheaper testing will be.
- Analysis and design also focuses on what data will be required in order to implement the tests as well as what the expected results of executing the tests.

The Process of Testing

Implementation and execution: Realizing the test cases and setting up the environment, e.g. test harness and automated test scripts, within which tests can be efficiently executed and results can be logged.

Evaluation: Making a judgement as to whether or not the exit/success criteria determined during the planning phase has been achieved, e.g. 85% of all executable statements within the code base have be executed.

Note: In the main, the activities described above represents a sequence. However, problems encountered in later phases may require earlier phases to be revisited.

Testing and Trace-ability

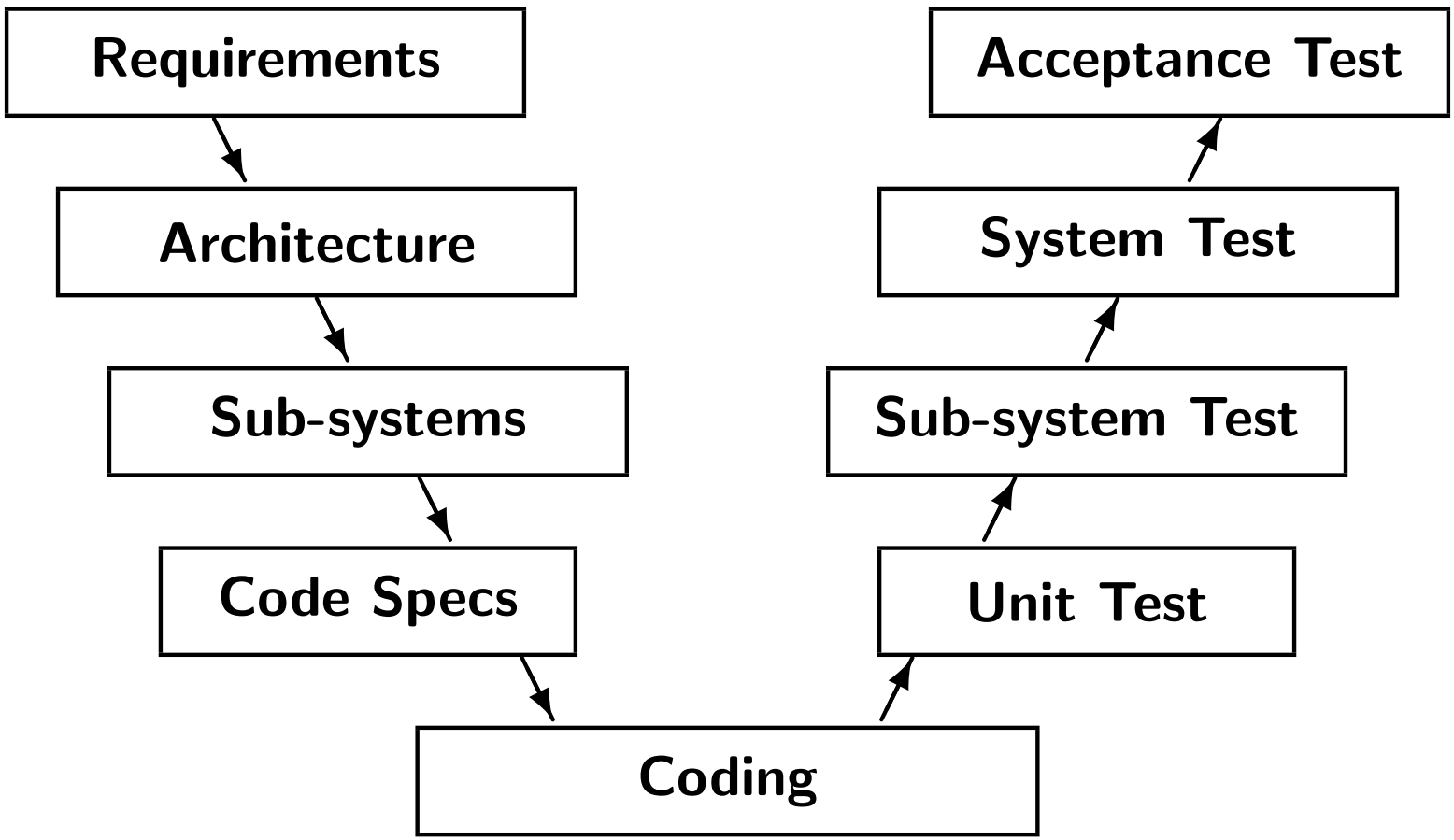
Requirement	**Sub-system**	**Module**	**Code**	**Tests**

reverse-thruster activation conditional on landing gear deployment	Avionics controller	`EngineCtrl`
		
		`BrakeCtrl`
			Lines 100,239
			
			Lines 52,123
				99,101
				
				11,51

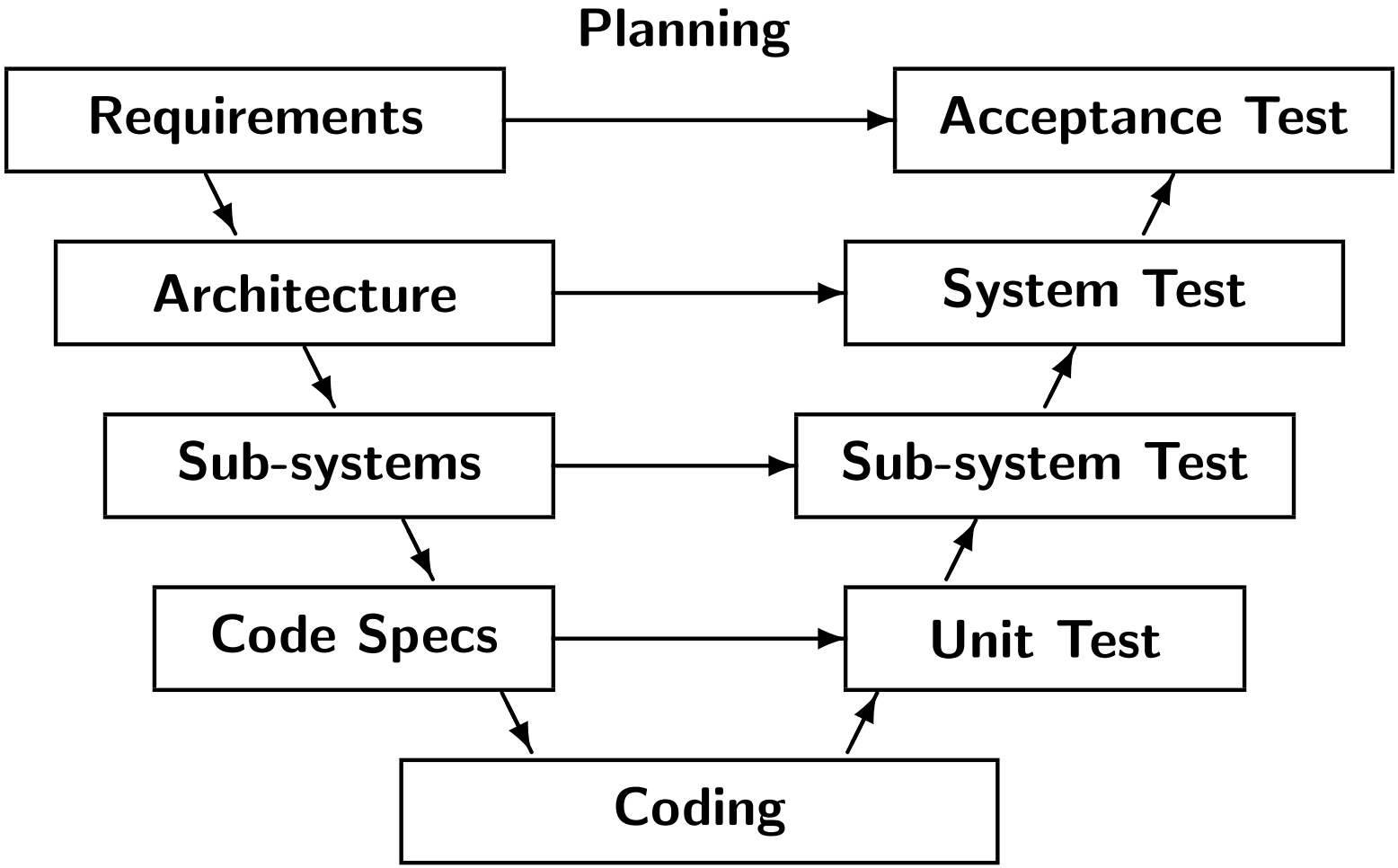
...	...	...	...	...

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

The V-Model for Software Development



Planning for Testing



Requirements Analysis

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?

Aside: Before requirements analysis, a Feasibility Study will have been undertaken, *e.g.* is the project cost-effective from a business perspective? Are the right skills and technologies available for the project? Information Systems students will pursue this thread during weeks 8, 9 and 10.

Requirements Analysis

- 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 analysis:
  - "Walk-throughs" or Fagan-style inspections – a formal style of meeting where specific tasks relating to quality are undertaken, *e.g.* bug detection.
  - 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.

Design Analysis

- 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.
- Aside: 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.

Unit Testing

- Unit testing is concerned with the low-level structure of program code. A *unit* might be a function or procedure. Within object-oriented programming the smallest executable unit is an object of a class – although within an object a number of methods may exist.
- Unit tests aim to test whether the code achieves its associated specification. In addition, unit testing should aim to identify redundant code.
- Effective unit testing requires a mechanized framework, e.g. JUnit, AUnit, PyUnit, ...
- Test Driven Development is an evolutionary approach to code level development that is led by test case design, i.e. write the tests then develop the code.

Aside: Unit testing techniques will be the focus of static & dynamic analysis lectures (Computer Science thread: weeks 8, 9 & 10).

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 (static analysis).

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.

Summary

- The analysis & testing life-cycle.
- Prevention better than cure – analysis & testing should start early both in terms of immediate analysis and planning for testing – developing test cases helps clarify requirements.
- Planning is crucial given the time-limited nature of the testing activity – planning and test case generation should be, as far as possible, integrated within your design notations and formalisms.

References

- "Software Testing: An ISTQB-ISEB Foundation Guide (Second Edition). BCS 2010.
- "Test Driven Development: By Example", K. Bent, Addison-Wesley, 2002.
- "Testing Object-Oriented Systems", R.V. Binder, Addison Wesley, 1999.
- "Software Testing in the Real World", E. Kit, Addison-Wesley, 1995.
- "The Object Constraint Language: precise modeling with UML", J. Warmer & A. Kleppe, Addison-Wesley, 1998.
- IEEE Standard for Software Verification and Validation Plans, 1992 (IEEE/ANSI Std 1012-1986)
- IEEE Standard for Software Test Documentation, 1991 (IEEE/ANSI Std 829-1983)
- "The Art of Software Testing", Myers, G.J. Wiley & Sons, 1979.