Systems Engineering

Lecture 7

System Verification

Dr. Joanna Bryson
Dr. Leon Watts

University of Bath
Department of Computer Science

Learning outcomes

- After both lectures and doing the reading, you should be able to:
  - Describe in detail the purpose, scope of, and activities comprising each of the three main phases of software testing.
  - Discuss the THERAC-25 case study as a motivator for correct V&V practice.
  - Explain what is meant by a test coverage metric.
  - Describe the role of automated test tools in the three main phases of software testing.
  - Explain the role of debugging in the software process.

Software System Verification

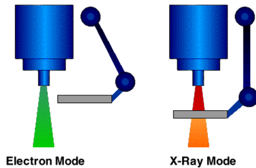
- "Doing the job right"
  - Static (Software Inspections)
  - Dynamic (Software Testing)
- Testing:
  - The process of exercising or evaluating a system by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results [IEEE 1983]
- Testing takes time and effort (~60% of development time?)

Specification & Verification

- Testing is only meaningful with a specification.
  - Better specification = better testing
  - Clear expectations
  - Prioritisation
  - Success metrics
  - Scenarios / Use cases

THERAC-25: Case Study

- THEraputic RAdiation Computer (1976)
  - Poor software testing practice causes harm & loss of life.



  - "Malfunction 54" – implied no dose when dose given.
  - Triggered by a certain (improbable?) key sequence.
  - One man 'team' (implementation and testing).
  - V&V considerations?

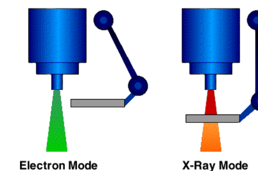
Testing

- Testing against a specification
  - Check for valid behaviour against requirements (positive).
  - Check for defects / non-specified behaviour (negative).
    - How do you think of every possible unspecified behaviour?
  - Effective testing requires both positive and negative tests.
- Testing strategies concerning code visibility:
  - Black box (Functional) Use the interface only.
  - White box (Structural) Use "insider knowledge" to check every possible path through the code.

Testing

- Have I tested enough?
  - No.
  - There will always be bugs, your organisation must deal with this.
  - However, at some point you have to release.
  - That point should be well-defined, for ethical & legal reasons.
- Testing against specification / Fitness for purpose.
- Test coverage metrics (whitebox.)
- Testing by experts vs. testing by novices:
  - quantity of testers matters more than quality!

THERAC-25: Case Study



- What makes this a good & a bad case study?

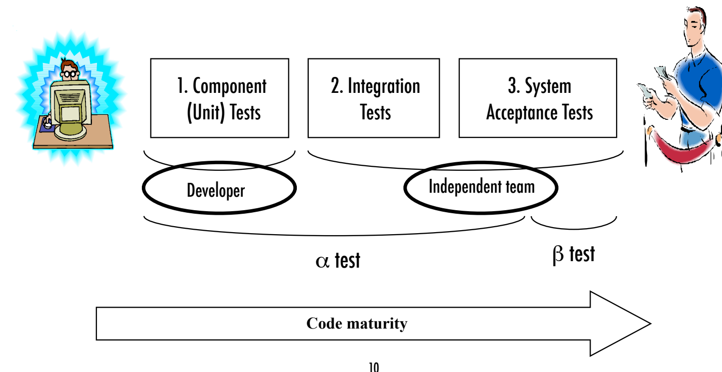
"Testing can only show the presence of errors, not their absence." [Dijkstra, 1972]

Qualities for Software Test Engineers

- Having a serious 'test to break' attitude,
  - a strong desire for quality and attention to detail.
- Balance stakeholder interests:
  - External stakeholder: the viewpoint of users, clients.
  - Internal stakeholders: tact and diplomacy to maintain a cooperative relationship with development team.
- Communication skills.
  - Testers serve at the interface between technical (developers) and non-technical (customers, management) people.
- Ability to judge high-risk areas of an application. *Experience*
  - Necessary to focus testing efforts.

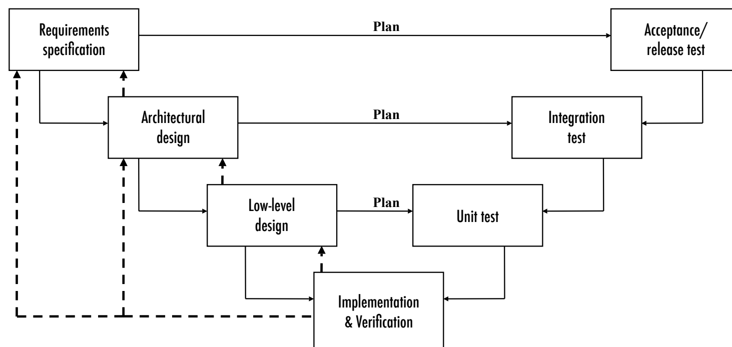
The Three Phases of software testing

- Each of the three main phases of testing vary in system and organisational scope.



V-Model: Planning for Verification

- The verification activity typically runs throughout the software lifecycle.



1. Component (Unit) Testing

- The lowest level of formal testing.
  - A unit is the smallest component provided with a specification.
  - Focus on functional requirements (and coverage metrics).
  - Usually a white box process.
- Each unit is tested in isolation.
  - Stub out external calls (within reason). Write drivers.
  - Can be a lot of work. CASE can help.
  - Advantages:
    - Easier to test a unit thoroughly when in isolation.
    - Can test many units in parallel.

1. Component (Unit) Testing

- OO, networking & concurrency can introduce additional complexity.
  - State space
  - Temporal dependency

Automated Testing

- Devising test cases can never be automated.
  - But running (many of) them can be...
- Test harnesses can provide automated:
  - Environment generators / cleanup,
  - Test selection and execution,
  - Test results analysis,
  - Report generation and archiving.
- Test harnesses are often built into configuration management software.

Test Harnesses

- Test harnesses usually take the form of scripts written by the development or test team.
- Two main parts:
  - Test execution engine
  - Test script repository

JUnit

```
public class MyTest extends TestCase {
   public void checkSquare() {
      Squarer s=new Squarer(3);
      Assert.assertTrue(s.result==9);
      // assertEquals(x,y) also used
   }
   …
}
```

- Java CASE tool that encourages test-first development.
  - Often used in agile methods e.g. XP
  - Creates test harness for units
  - Assists unit and regression testing
- Define tests by extending TestCase

JUnit

```
public class MyTest extends TestCase {
   public void checkSquare() {
      Squarer s=new Squarer(3);
      Assert.assertTrue(s.result==9);
      // assertEquals(x,y) also used
   }
   public static Test suite() {
      TestSuite suite=new TestSuite();
      suite.addTest(new MyTest("checkSquare()";
      ...
      return suite;
   }
}
```

- Specify a test harness as a 'suite'

JUnit

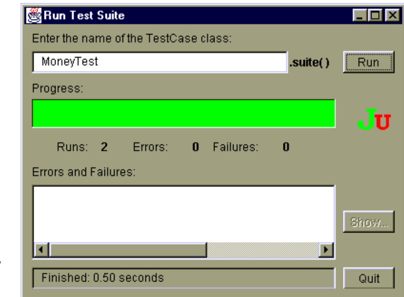


Image: sourceforge.net

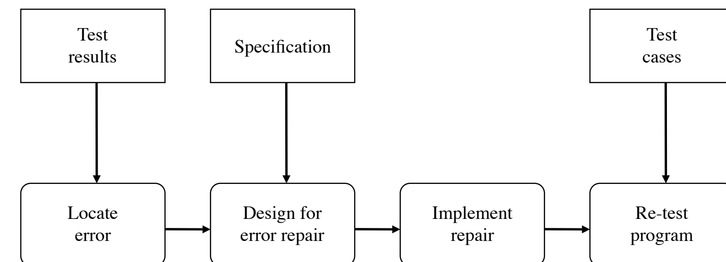
- Specify a test harness as a 'suite'
  - Suite runs in JUnit GUI.
- Good practice
  - Re-run your tests at least once a day during development e.g. lunch.
  - Keep updating your TestCase(s) as you develop.

Test Coverage Metrics

- What proportion or percentage of the software have we tested?
  - Provides a measure of confidence in the rigor of our testing process.
- Common coverage metrics are: *Whitebox*
  - Statement coverage: % of executable statements run at least once.
  - Decision coverage: % of decision outcomes (branches) run at least once.
  - Condition coverage: % of conditions causing decisions independently tested.

Debugging Process

- A typical debugging process when *testing*:



Debugging Process

- Debugging, like programming, is an art learnt primarily through experience.
  - Learn common errors (more about this later) and spot patterns.
- Interactive Development Environments (IDEs) help debugging.
  - Access to symbol table in compiler.
  - Allow developer to step through code
    - Set breakpoints.
    - Add variable watches.

Can you do this in Eclipse?

- When debugging design documentation: some tools are also available e.g. Rational Rose.

Common Program Defects

- There are two classes of common defect:
  - General issues, regardless of particular language,
  - Program language specific.
- Over to you...
  - Can you think of any **general** errors you see (or make!) in programs time and time again?

General program defects (1)

- Data faults
  - Variables initialised before they are used?
  - All constants named?
  - Bounds
  - String delimiters
  - Compound statements (scope)
- Control faults
  - All cases accounted for in case constructs? Breaks if needed?
  - Predicate logic on conditional statements.
  - Compound statements (control).
- Exception management
  - All possible (reasonable) error conditions handled?

General program defects (2)

- Interface faults
  - Number/type/order of arguments match?
  - Shared data structures.
- Memory
  - Has enough (any?!) memory been allocated? (buffer overflow)
  - Memory leaks (slows execution even if garbage collection).
  - Stack overflow (end cases for recursion).
  - Pointer arithmetic.

Error Seeding / Bebugging

- Intentionally introduce "typical" defects into the code.

$$\frac{Seeded\ detected\ (SD)}{Seeded\ (S)} = \frac{Unseeded\ detected\ (UD)}{Unseeded\ (U)}$$

- Example:
  - TD = total detected = (SD+UD) = 40
  - S = 100 (known)
  - SD = 5 (how many seeded errors were found)
  - U = estimated total unseeded = 100 (40 - 5) / 5 = 700

Defensive Programming

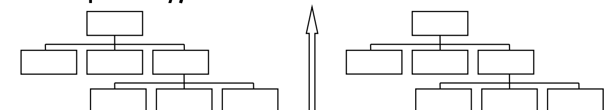
- Tests can be built into the code for execution at run-time.
  - E.g. range checks.
  - Heavily used in safety critical systems to prevent unsafe values
  - e.g. high doses of radiation.
- These usually take the form of "assertions"
  - `assert(x<3)`
  - "I assert `x` should be `<3` at this point. If not (if a false assertion), then throw a fatal error".
  - Indicates system aborting is more desirable than the consequence of a value of x $\geq$3.

2. Integration Testing

- Systems are built from components. These components must be integrated at some stage to form the system.
- The correct interaction of components must be verified.
  - Typically a white box process.
  - Iterate back to developers for debugging.
- Prioritise the integration of components.
- Implications for Testing:
  - i. Top-down vs. ii. Bottom-up integration
  - iii. Regression testing

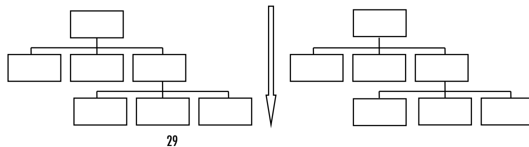
i. Bottom-up integration & testing

- Benefits:
  - No stubs needed.
  - Provides early integration of units.
  - Overall structural design can evolve until a late stage.
- Problems:
  - Need an initial fully decomposed design to start (testing and design cannot overlap initially).



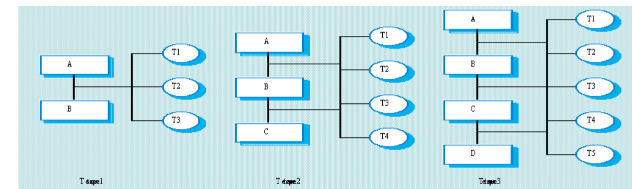
ii. Top-down integration & testing

- Achieved by stubbing out lower functions.
  - Gives a good progress indication of overall functionality.
- Many stubs need to be written.
  - Stubbing may be seen as expensive for large projects.
  - Benefits may outweigh the costs.
  - Stubs can be seen as specifications.



iii. Regression testing

- When new components are integrated...
  - We must test their integration with connected units, but
  - must also ensure the existing inter-operation of units is not damaged.
- ...therefore we employ regression testing.
  - Automation is necessary on medium to large scale projects.

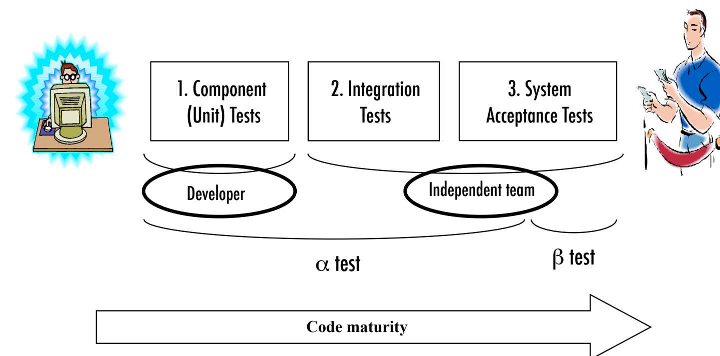


3. Release / Acceptance Testing

- Testing the system as a functioning entity that might be delivered to the customer.
  - Black box testing.
  - Based on both Functional & Non-Functional Requirements.
- Typically test cases will include:
  - Scenarios / Use-case-based tests
  - Equivalence partitioning.
  - Boundary value testing.
  - Walking the state transition table.
  - Stress testing and other external measures.

The Three Phases of software testing

- Each of the three main phases of testing vary in system and organisational scope.



Summary

- After both lectures and doing the reading, you should be able to:
  - Describe in detail the purpose, scope of, and activities comprising each of the three main phases of software testing.
  - Discuss the THERAC-25 case study as a motivator for correct V&V practice.
  - Explain what is meant by a test coverage metric.
  - Describe the role of automated test tools in the three main phases of software testing.
  - Explain the role of debugging in the software process.

Whitebox Testing Tips

- Test against subsystem functional specification.
- Typical defect tests would include code centred:
  - Equivalence partition / boundary checks
  - Tests of branches
  - Tests of conditions
  - Concurrency/timing and contended resource issues