Systems Engineering

Lecture 9

System Verification II

Dr. Joanna Bryson
Dr. Leon Watts

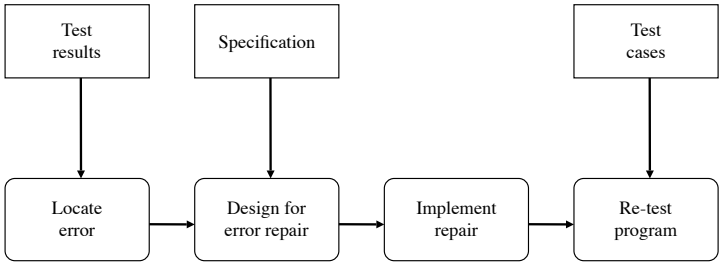
University of Bath
Department of Computer Science

Learning outcomes

- After both lectures and doing the reading, you should be able to:
  - Explain the role & practice of debugging in the software process.
  - Describe in detail the purpose, scope of, and activities comprising each of the three main phases of software testing.
  - Conduct a software inspection / code review.
  - Describe the components of a software test case and explain its connection with system requirements.

Debugging Process

- A typical debugging process when *testing*:



Debugging Process

Review

- Debugging, like programming, is an art learnt primarily through experience.
  - Learn common errors 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.

You should be able to do this in Eclipse.

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

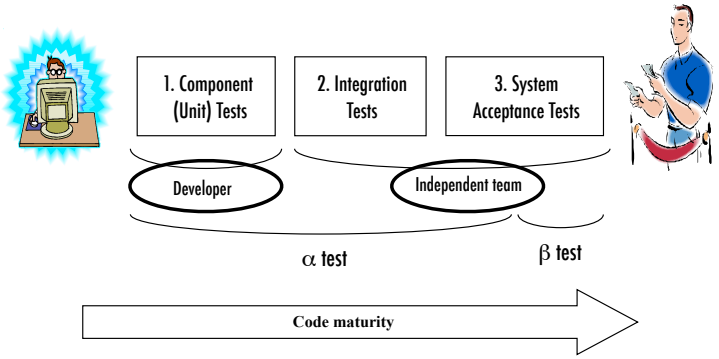
Review?

Defensive Programming

- Tests can be built into the code for execution at run-time.
  - E.g. range checks, "impossible" default options for switch statements.
  - Heavily used in safety critical systems to prevent unsafe values, e.g. high doses of radiation.
  - Remember: not just checking for yourself, checking future modifications.
- 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 I throw a fatal error."
  - Indicates system aborting is more desirable than the consequence of a value of x ≥3.

The Three Phases of software testing

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



1. JUnit

Review: Component

```
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

1. JUnit

Review: Component

```
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'

1. JUnit

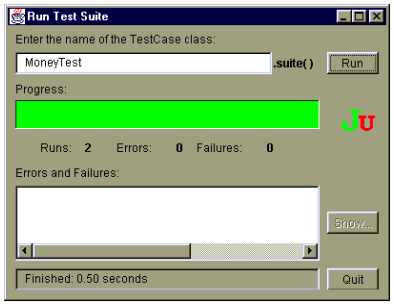


Image: sourceforge.net

- 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.

Review: Component

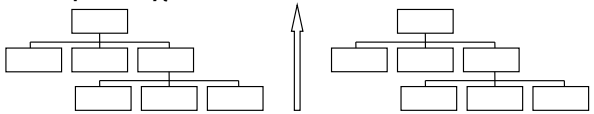
Whitebox Testing Tips

- 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

2. Bottom-up integration & testing

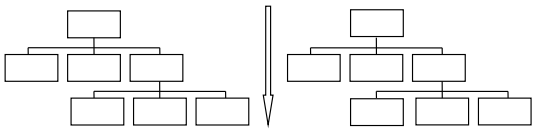
Review: Integration

- 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).



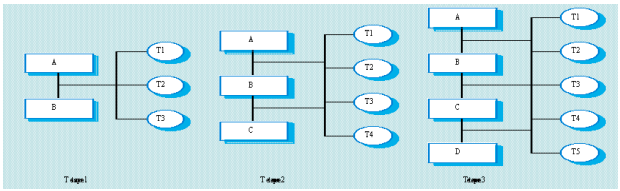
2. Top-down integration & testing

- 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.



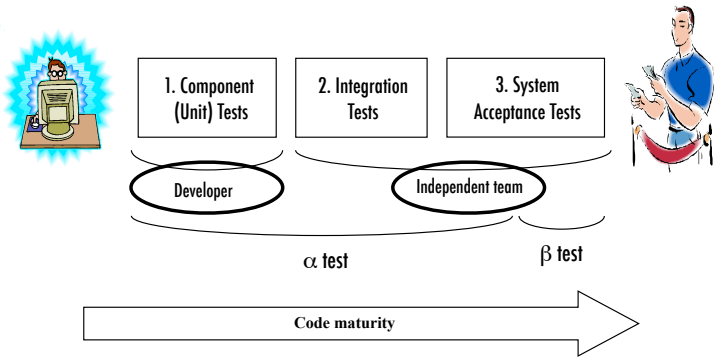
3. Release / Acceptance Testing

- 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.

Review: Acceptance

The Three Phases of software testing

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



Software Inspections

- Systematic review of software documentation
  - Usually program code, but possible with any readable system documentation.
  - Performed in the setting of a group meeting.
- There are 3 significant advantages to inspections:
  - Program need not be complete in order to inspect.
  - One error can mask others at run-time. Because inspection is static, this is not an issue and more errors may be picked up.
  - Can consider broader issues e.g. coding standards, portability, maintainability.

How to run a Software Inspection

- **Create checklist before meeting.**
- System overview presented to inspection team at a meeting. (500 statements/hour)
- Code / documentation distributed to inspection team in advance. (150 statements/hour)
- Inspection takes place in group meeting. Errors are noted. (100-150 statements/hour)
- Code / documentation modified (debugged).
- Re-inspect? (decision based on V&V budget)

Roles in a Software Inspection

- Reader
  - Presents code to the meeting, sometimes stepping through lines.
- Inspector
  - Spots the errors, or non-compliances with broader issues such as standards.
- Author / Owner
  - Responsible for producing the code or document being reviewed.

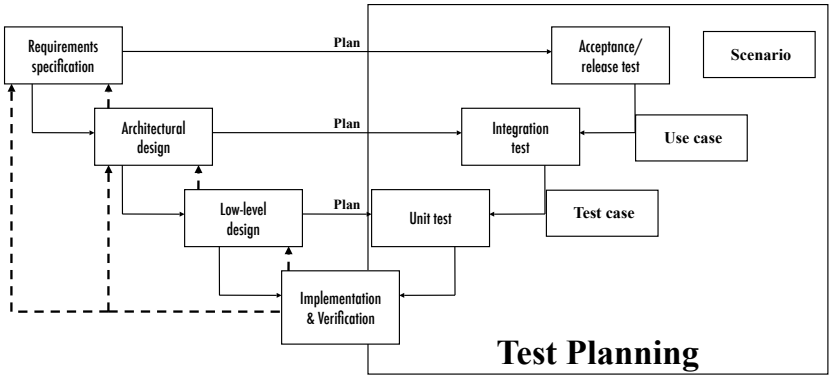
- Scribe
  - Records results of inspection meeting.
- Chairman / Moderator
  - Responsible for running inspection.
- Chief moderator **QA**
  - Reflects on / improves inspection process itself

Features of Software Inspections

- Formal methods may be used for rigorous software inspections e.g. Cleanroom.
- Informal methods tend to focus on:
  - Defect detection,
  - Poor programming style,
  - Standards conformance.
- Informal inspections can detect up to 60% of defects [Fagan '86].
  - When applying formal methods, this can rise to 90% [Mills et al. '97].
- Particularly valuable for loose-typed languages such as C.
  - Less useful for languages like Java as compiler does more of the work.
- They are expensive, and become cost effective only when inspection teams are experienced.

Effective Testing in the V-Model

Joined-up Thinking about Validation and Verification: Test Cases



How to Design Test Cases

- What is a test case?
  - "A set of inputs and predicted outputs that are effective in discovering program defects and showing the system meets its requirements".
- Three approaches to designing test cases
  - Requirement based
  - Partition based
    - "equivalence partition"
  - Structure / path based

Increasingly low-level

- Exhaustive testing of a system is impossible in all but the simplest cases. Recall: fitness for purpose.

Writing Effective Test Cases

- A test case should contain:
  1. a clear statement of its objective, joined up with other development documents, especially requirements.
  2. test case name (meaningful title).
  3. test case identifier (unique code number).
  4. description of the test conditions/setup.
  5. input data requirements.
  6. the steps to be performed with the test setup.
  7. the results of the test that are expected, if all is working properly.

Summary

- After both lectures and doing the reading, you should be able to:
  - Explain the role & practice of debugging in the software process.
  - Describe in detail the purpose, scope of, and activities comprising each of the three main phases of software testing.
  - Conduct a software inspection / code review.
  - Describe the components of a software test case and explain its connection with system requirements.