CS 619 Introduction to OO Design and Development

Testing

Fall 2012

Overview

- Preliminaries
- All sorts of test techniques
- Comparison of test techniques
- Software reliability

- Main issues:

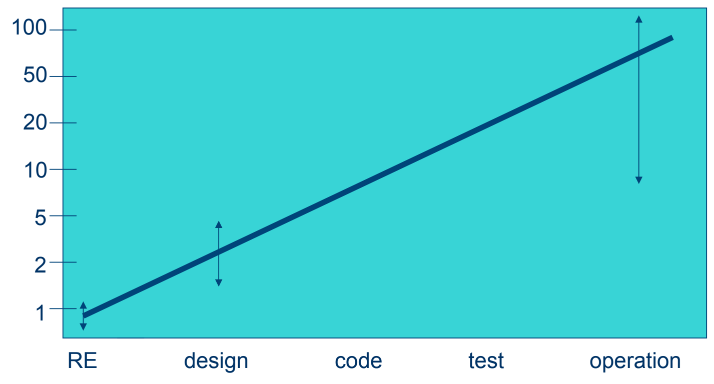
There are a great many testing techniques

Often, only the final code is tested

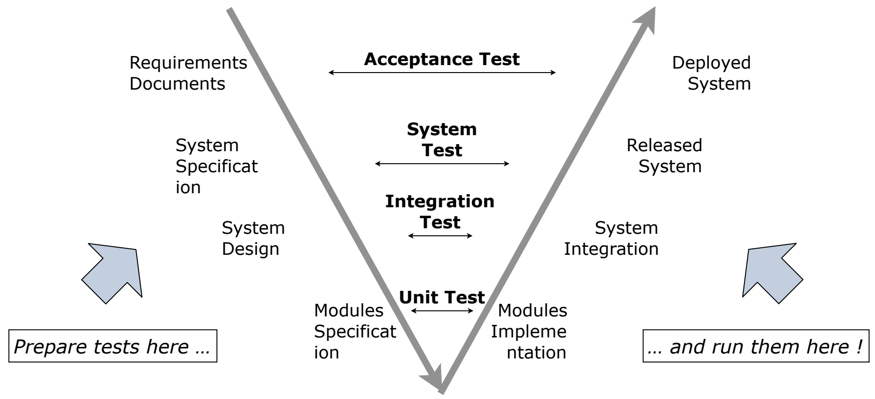
State-of-the-Art

- 30-85 errors are made per 1000 lines of source code
- extensively tested software contains 0.5-3 errors per 1000 lines of source code
- testing is postponed, as a consequence: the later an error is discovered, the more it costs to fix it.
- error distribution: 60% design, 40% implementation.
- 66% of the design errors are not discovered until the software has become operational.

Relative cost of error correction

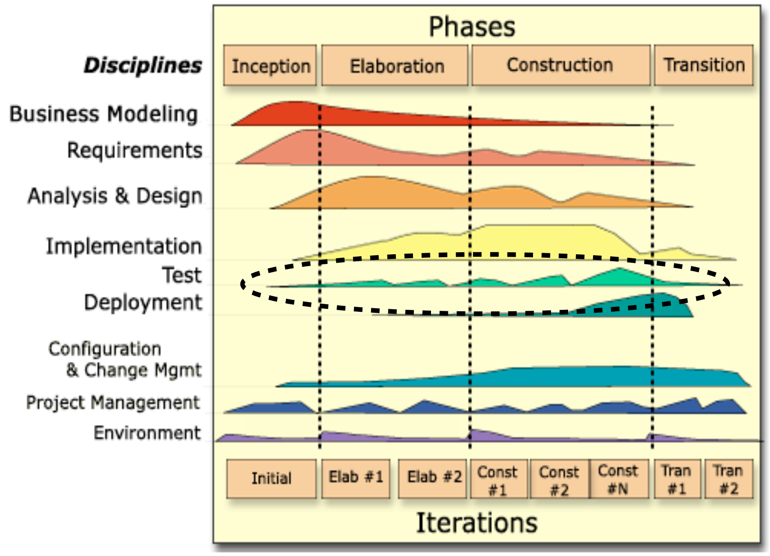


When to Test: the V-model



SPECIFY & DESIGN WITH TESTABILITY IN MIND

When to Test: UP Process



Program testing

- Can reveal the presence of errors NOT their absence
  - Only exhaustive testing can show a program is free from defects. However, exhaustive testing for any but trivial programs is impossible
- A successful test is a test which discovers one or more errors
- Should be used in conjunction with static verification
- Run all tests after modifying a system

What is Testing

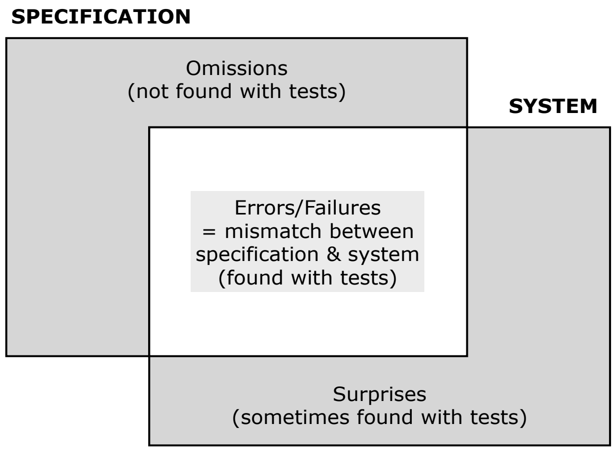
Testing should

- *verify* the requirements (Are we building the product right?)
- NOT *validate* the requirements (Are we building the right product?)

Definitions

- Testing
  - + Testing is the activity of executing a program with the intent of finding a defect
    - - A successful test is one that finds defects!
- Testing Techniques
  - + Techniques with a high probability of finding an as yet undiscovered mistake
    - - Criterion: *Coverage* of the code/specification/requirements/…
- Testing Strategies
  - + Tell you *when* you should perform *which* testing technique
    - - Criterion: *Confidence* that you can safely proceed
    - - Next activity = other testing until deployment

What is Testing



Some Terminology: error, fault, failure

- an *error* is a human activity resulting in software containing a fault
- a *fault* is the manifestation of an error
- a *fault* may result in a failure

More Terminology

Component
- A part of the system that can be isolated for testing
  - + an object, a group of objects, one or more subsystems

Test Case
- A set of inputs and expected results that exercise a component with the purpose of causing errors and failures
  - + a predicate method that answers "true" when the component answers with the expected results for the given input and "false" otherwise
    - - "expected results" includes exceptions, error codes,...

Test Stub
- A partial implementation of components on which the tested component depends
  - + dummy code that provides the necessary input values and behaviour to run the test cases

Test Driver
- A partial implementation of a component that depends on the tested component
  - + a "main()" function that executes a number of test cases

Classification of testing techniques

- Classification based on the source of information to derive test cases:
  - black-box testing (functional, specification-based)
  - white-box testing (structural, program-based)
- Classification based on the criterion to measure the adequacy of a set of test cases:
  - coverage-based testing
  - fault-based testing
  - error-based testing

Black-box Testing

- A.K.A Functional Test, Test in Large
- Focus: I/O behavior. If for any given input, we can predict the output, then the module passes the test.
  - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.

  E.g. If an object is supposed to accept a negative number, testing one negative number is enough.

Equivalence Partitioning : Example

Example: Binary search

```
private int[] _elements;
public boolean find(int key) { ... }
•pre-condition(s)
    - Array has at least one element
    - Array is sorted
•post-condition(s)
    (The element is in _elements and the result is true)
    or (The element is not in _elements and the result is false)
```

Check input partitions:
- Do the inputs satisfy the pre-conditions?
- Is the key in the array?

  ➡ leads to (at least) 2x2 equivalence classes

Check boundary conditions
- Is the array of length 1 ?
- Is the key at the start or end of the array?

  ➡ leads to further subdivisions
  (not all combinations make sense)

Equivalence Partitioning: Test Data

Generate test data that cover all meaningful equivalence partitions.

Test Cases	**Input**	**Output**
Array length 0	key = 17, elements = { }	FALSE
Array not sorted	key = 17, elements = { 33, 20, 17, 18 }	exception
Array size 1, key in array	key = 17, elements = { 17 }	TRUE
Array size 1, key not in array	key = 0, elements = { 17 }	FALSE
Array size > 1, key is first element	key = 17, elements = { 17, 18, 20, 33 }	TRUE
Array size > 1, key is last element	key = 33, elements = { 17, 18, 20, 33 }	TRUE
Array size > 1, key is in middle	key = 20, elements = { 17, 18, 20, 33 }	TRUE
Array size > 1, key not in array	key = 50, elements = { 17, 18, 20, 33 }	FALSE
…	…	…

White-box Testing

- A.K.A. Structural testing, Testing in the small
- Treat a component as a “white box”, i.e. you can inspect its internal structure
- Internal structure is also design specs; e.g. sequence diagrams, state charts, ...
- Derive test cases to maximize coverage of that structure, yet minimize number of test cases

White-box Testing Coverage Criteria

- Statement Testing: Test single statements
- Loop Testing:
  - Cause execution of the loop to be skipped completely. (Exception: Repeat loops)
  - Loop to be executed exactly once
  - Loop to be executed more than once
- Path testing:
  - Make sure all paths in the program are executed
- Branch Testing (Conditional Testing): Make sure that each possible outcome from a condition is tested at least once

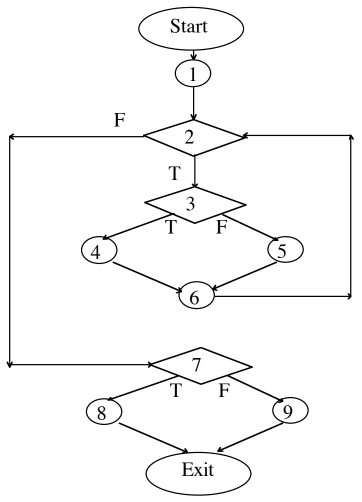
```
if ( i =  TRUE) printf("YES\n");        else  printf("NO\n");
Test cases: 1) i = TRUE; 2) i = FALSE
```

White-box Testing: Determining the Paths

```
    FindMean (FILE ScoreFile)
    {  float SumOfScores = 0.0;
       int NumberOfScores = 0;
       float Mean=0.0; float Score;                          1
       Read(ScoreFile, Score);
  2    while (! EOF(ScoreFile) {
     3    if (Score  > 0.0 ) {
                SumOfScores = SumOfScores + Score;           4
                NumberOfScores++;
          5   }

             Read(ScoreFile, Score);                         6
       }
       /* Compute the mean and print the result */
  7    if (NumberOfScores > 0) {
               Mean = SumOfScores / NumberOfScores;          8
               printf(" The mean score is %f\n", Mean);
       } else
               printf ("No scores found in file\n");         9
    }
```

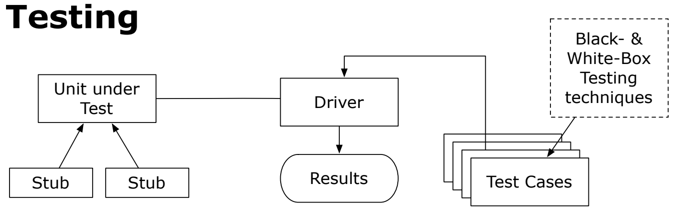
Constructing the Logic Flow Diagram



Testing stages

- Unit testing
  - Testing of individual components
- Integration testing
  - Testing to expose problems arising from the combination of components
- System testing
  - Testing the complete system prior to delivery
- Acceptance testing
  - Testing by users to check that the system satisfies
    requirements. Sometimes called alpha testing

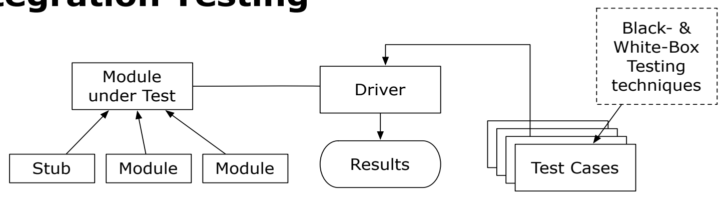
Unit Testing



- Why?
  + Locate small errors (= within a unit) fast
- Who?
  + Person developing the unit writes the tests.
- When ? At the latest when a unit is delivered to the rest of the team
  + No test ⇒ no unit
  + Test drivers & stubs are part of the system ⇒ configuration management

*** **Write the test first,**
- i.e. before writing the unit.
- It will encourage you to design the component interface right

Integration Testing



- Why ?
  + The sum is more then its parts, i.e. interfaces (and calls to them) may contain defects too.
- Who ?
  + Person developing the module writes the tests.
- When ?
  + Top-down: main module before constituting modules
  + Bottom-up: constituting modules before integrated module
  + In practice: a little bit of both

## The distinction between unit testing and integration testing is not that sharp!

Regression Testing

Regression Testing ensures that all things that used to work still work after changes.

Regression Test
- *= re-execution of some subset of tests to ensure that changes have not caused unintended side effects*
- tests must avoid regression (= degradation of results)
- Regression tests must be repeated often (after every change, every night, with each new unit, with each fix,...)
- Regression tests may be conducted manually
  + Execution of crucial scenarios with verification of results
  + Manual test process is slow and cumbersome
    ➡ preferably completely automated

Advantages
- Helps during iterative and incremental development
  + during maintenance

Disadvantage
- Up front investment in maintainability is difficult to sell to the customer

Acceptance Testing

Acceptance Tests
- conducted by the end-user (representatives)
- check whether requirements are correctly implemented
  + borderline between verification (“Are we building the system right?”) and validation (“Are we building the right system?”)

Alpha- & Beta Tests
- acceptance tests for “off-the-shelves” software (many unidentified users)
  + Alpha Testing
    - end-users are invited at the developer’s site
    - testing is done in a controlled environment
  + Beta Testing
    - software is released to selected customers
    - testing is done in “real world” setting, without developers present

More Testing Strategies

Recovery Testing
- Test forces system to fail and checks whether it recovers properly
  - + For fault tolerant systems

Stress Testing (Overload Testing)
- Tests extreme conditions
  - + e.g., supply input data twice as fast and check whether system fails

Performance Testing
- Tests run-time performance of system
  - + e.g., time consumption, memory consumption
    - ➡ first do it, then do it right, then do it fast

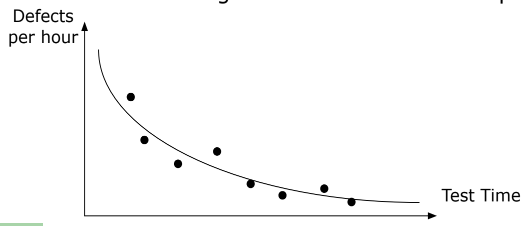
Back-to-Back Testing
- Compare test results from two different versions of the system
  - + requires N-version programming or prototypes

When to Stop ?

When are we done testing? When do we have enough tests?

Cynical Answers (sad but true)
- You're never done: each run of the system is a new test
  - ➡ Each bug-fix should be accompanied by a new regression test
- You're done when you are out of time/money
  - ➡ Include test in project plan
    AND DO NOT GIVE IN TO PRESSURE
  - ➡ ... in the long run, tests SAVE time

Statistical Testing
- Test until you've reduced failure rate under risk threshold
  - ➡ Testing is like an insurance company calculating risks



Unit-testing Heuristics

1. Create unit tests as soon as object design is completed:
   - Black-box test: Test the use cases & functional model
   - White-box test: Test the dynamic model
   - Data-structure test: Test the object model
2. Develop the test cases
   - Goal: Find the minimal number of test cases to cover as many paths as possible
3. Cross-check the test cases to eliminate duplicates
   - Don't waste your time!
4. Desk check your source code
   - Reduces testing time
5. Create a test harness
   - Test drivers and test stubs are needed for integration testing
6. Describe the test oracle
   - Often the result of the first successfully executed test
7. Execute the test cases
   - Don't forget regression testing
   - Re-execute test cases every time a change is made.
8. Compare the results of the test with the test oracle
   - Automate as much as possible

Test-Driven Development (TDD)

- First write the tests, then do the design/ implementation
- Part of agile approaches like XP
- Supported by tools, e.g. JUnit
- Is more than a mere test technique; it subsumes part of the design work

Steps of TDD

1. Add a test
2. Run all tests, and see that the system fails
3. Make a small change to make the test work
4. Run all tests again, and see they all run properly
5. Refactor the system to improve its design and remove redundancies