

- Testing
 - Chapter 23
 - Chapter 22
 - testplan.doc

Definitions

- *Verification*: has the program been implemented correctly?
 - "Are we building the program right?"
 - Process question
- *Validation*: has the correct program been implemented?
 - "Are we building the right program?"
 - Product question
- *Testing*: the process of executing a program with the intent of detecting problems
 - Can be used for either verification or validation

- *Error*: a (human) mistake made in the process of constructing software
- *Fault/bug/defect*: the manifestation of an error in the software
- *Failure*: a condition arising when executing software produces an incorrect result

- - α (*alpha*) test: informal, in-house, early
- - β (*beta*) test: formal, out-of-house, pre-release
- *Sanity check*: quick-and-dirty, just before shipping

Approaches to Verification

- Testing: exercising software to try and generate failures
- Inspection/review/walkthrough: systematic group review of program text to detect faults
- Formal proof: proving that the program text implements the program specification
- (execution vs. non-execution based testing)

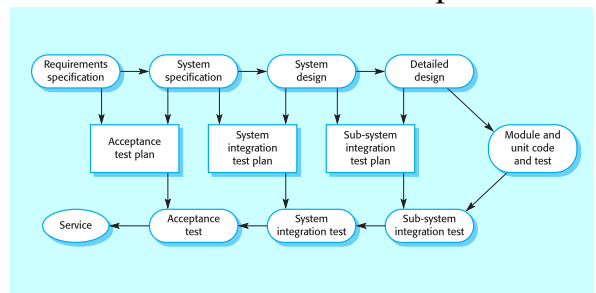
Comparison

- Testing:
 - purpose: create failure
 - danger: inadequate test set
- Proof
 - p: prove correctness
 - d: only/select functionality
- Review
 - p: detect defects
 - d: informal
- Debugging
 - p: find and correct defects
 - d: local fixes and new defects

Testing and S/W Lifecycle

Requirements Acceptance Testing
 Design Integration Testing
 Coding Unit Testing
 Maintenance Regression Testing

The V-model of development



Adequacy criterion

- Test sets, like programs, are designed, to accomplish a goal—to adequately exercise a program
- *Adequacy criterion*: precise, deterministic rule for deciding beforehand how you will know when to stop testing
 - Alternatively—how do you know when you have a good suite of tests?
- Examples
 - Every statement executed
 - Every requirement tested
 - Error-based tests
 - Reliability level

Test Planning

- Adequacy criterion
- For each test
 - What is the objective of the test?
 - How will you know if the objective has been met?
 - The input to use
 - Expected output
 - The actual output obtained

Test Organization

- *Top down*: test the main routine plus stubs; then replace a stub by code (+ more stubs) and integrate
 - Advantage: always have a complete system
 - Disadvantages: stub generation; meaningless output
- *Bottom up*: test modules separately, then combine
 - Advantages: meaningful values generated
 - Disadvantages: hides integration problems until late
- *Incremental*: try to add one piece at a time so that you know what to concentrate on if a failure occurs

Types of Testing

- *Black box (functional)*: look only at the behavior of the program
- *White box (clear box)*: base the testing on the structure of the code

Black Box

Generate tests to check desired program functionality without knowledge of how the code works

- *Equivalence partitioning*: divide possible test sets into equivalence classes (for valid and invalid inputs), and run one test from each class
- *Boundary value analysis*: look for test cases on the boundaries of the equivalence classes
- *Exhaustive testing*: run every possible input (infeasible)
- *Functional testing*: construct tests directly from the requirements document
- *Random*: if the tests are randomly generated with a distribution that corresponds to the expected usage of the program, then you can get a reliability estimate, such as MTBF, to use as an adequacy criterion

White Box

- exhaustive not possible (in reality)
- alone not sufficient
 - forgets about specs
 - doesn't check if logic is flawed
 - are proper equations used?

White Box testing - Coverage

- Adequacy based on control or data flow properties
- *Statement coverage*: % of all executable statements exercised during testing
 - Testing a statement once does not guarantee that the statement is correct
 - Some statements may be "dead code"
- *Branch coverage*: % of conditional branches exercised
 - (do tests exercise both the `then` and `else` branches of each `if` statement even if one of them is empty?)

- *Condition coverage*: % of boolean expression constituents exercised
 - Was there a test that caused the `then` branch to be taken because `a` was true? Because `b` was true?
`if (a || b) then ... else ...`
- *Path coverage*: % of program paths executed (infeasible or impossible)
- *Data flow coverage* measures
 - For a given use of a variable and for each assignment to that variable that could "reach" that use, is there a test that causes execution to proceed from the assignment to the use?

Error-based testing

- Look for evidence of a specific class of errors, such as “off by one”, known to occur in programs
 - error checklists, (may be language-dependent)
- Adequacy is the extent of possible error opportunities evaluated
- Mutation analysis (fault seeding)
 - Generate a set of program variants (*mutants*) containing a comprehensive set of intentionally seeded bugs representative of typical errors
 - Compose tests sufficient to distinguish the generated programs from the original (kill the mutant)

non-functional testing/analysis

- Performance
- Load
- Usability
- Security
- Portability
- Maintainability / complexity metrics
- Coding standards

- you will maybe look at performance, load, reliability, usability...
- need operational model
- instrumentation of the code?
- still need quantifiable adequacy criteria – when are you done testing, when do you determine that system meets this non-functional requirement
- stress-testing valuable, even if conditions outside of operational model

Guidelines

- "Program testing can best show the presence of errors but never their absence" - Dijkstra
- 50% of program development effort goes into testing, primarily integration testing
- "The act of designing tests is one of the most effective error prevention mechanisms known" - Beizer

- Effective testing requires a different attitude (destructive) from effective programming (constructive)
- Test functional as well as non-functional requirements
- Test that a program does what it is supposed to do and doesn't do what it isn't supposed to do
- Test how well the program deals with erroneous input and internally detected failures

- Tests should be traceable to customer requirements
- Tests should be planned before testing begins
- 80% of defects will be traced to 20% of modules
- Exhaustive testing is not possible
- To be most effective, testing should be conducted by an independent third party

- Include Release Test
 - product deployment, installation, configuration...
- Regression testing
 - rerun tests after updates or integration
 - identify “critical” regression tests
 - man need to automate (scripts, testing tools, harness of tests and input/output data...)
 - one reason which makes maintenance costly
- Beware not to consider outputs resulting from previous test

Non-execution based Verification

- Statistics show reduces time of execution based testing
- Informal reviews:
 - desk checks – unstructured, driven by author upon e.g., minor changes to non-critical components
 - walkthroughs – more structure, involve a meeting, major change to non-critical component
- Formal reviews:
 - inspections: formal, well defined structure, assigned roles to participants, defined entry/exit criteria for phases in inspection process

Bug Tracking

- database w/ bug reports
- bug id, name/title, (keywords)
- area where it occurred, version number
- priority, (high, low... must fix for release x); severity
- conditions under which occurred, platform/environment, ... -> how to reproduce, what occurred vs. expected outcome
- status fixed, will not fix, duplicate, non-reproducible, by design...
- assigned to -> who should fix it
- don't overkill bug tracking interface – should be easy to use, so that it actually gets used!
- products (e.g., Bugzilla)

