

## Testing

Fritz Henglein  
DIKU

## Sources

- ✎ Glenford Myers, “The Art of Software Testing”, John Wiley & Sons, 1979
- ✎ Edward Kit, “Software Testing in the Real World”, Addison-Wesley, 1995
- ✎ Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, “Fundamentals of Software Engineering”, Prentice Hall, 1991

## Psychology of program testing

- ✎ **Wrong:** Testing has the purpose of documenting the correctness of a program
- ✎ **Correct:** Testing is the activity of executing programs to find errors in it.
- ✎ **Why?** Goal formulation impacts our focus. We are motivated by *success* (“yippie – bug!”) and demotivated by *failure* (“darn -- bug...”).

## Economics of testing

- ✎ Exhaustive testing is usually impossible or practically or economically infeasible
- ✎ The purpose of testing is improving software quality
- ✎ Finding (and eventually correcting) as many bugs as possible with few resources increases quality
- ✎ A *successful* test is one that identifies a bug (“yippie, bug!”), just like a successful medical test.

## Testing principles (basic)

- ✎ *Testing* consists of executing a program with the purpose of finding bugs in it.
- ✎ A *good* test case is one that gives high probability for uncovering a (new) bug.
- ✎ A *successful* test run is one that uncovers a new bug.

## Testing principles...

- ✎ Testing requires a *definition of the expected results (output data)* for individual test cases.
- ✎ A programmer should avoid testing his/her own program
- ✎ A programming organization should avoid testing its own programs
- ✎ Investigate the results of each test case very carefully

## Testing principles...

- ⌘ Test cases should be provided for both valid and expected inputs *and* for invalid and unexpected inputs
- ⌘ Testing that a program executes what it is expected to do is only half of the story. The other half is testing that it does *not* do something for the cases it is expected *not* to handle.

## Testing principles...

- ⌘ Make test cases repeatable. Do not throw them away.
- ⌘ Do not approach testing under the implicit assumption that no bugs will be found.
- ⌘ The probability of another bug found in a code fragment is proportional to the number of bugs already found in that fragment.

## Testing principles...

- ⌘ Testing is a highly creative and intellectually challenging task.

## Code inspections and walk-throughs

- ⌘ Group of 3-4 people going over code.
- ⌘ NB: At least 2-3 *outside* sets of eyeballs.
- ⌘ Good at not only asserting presence of bugs, but actually isolating *causes* for whole *bug series*.
- ⌘ Good at finding coding errors, not requirements or design errors.
- ⌘ Complementary to (dynamic) testing: some bugs easier found by code inspection/walk-through, some by testing.

## Code inspection

- ⌘ 4 persons, 1 acting as meeting leader (must not be author of program code under inspection)
- ⌘ Meeting leader keeps team focused (finding bugs, not fixing them)
- ⌘ Duration: typically 2 hours, without interruption
- ⌘ Procedure:
  - ⌘ Programmer explains program logic line-by-line. Group members ask questions, which must be answered. (Often bugs are found by programmer during reading "and then... ho, that can't be right.")
  - ⌘ Program is analyzed with respect to known categories of bugs (check list)

## Code inspection...

- ⌘ After meeting:
  - ⌘ programmer receives bug list for fixing
  - ⌘ leader may call another inspection
  - ⌘ bug list is used to update check list
- ⌘ Variation: fix bugs during inspection
- ⌘ Important: Must not be understood as "attack on author"

## Code walkthrough

- ≈ 3-5 persons, 1 acting as meeting leader, 1 as secretary
- ≈ 1 person (the “tester”, must not be author of code) comes with a set of *simple test cases*
- ≈ the test cases are executed manually, with program states noted on paper
- ≈ group members ask questions during this process
- ≈ after meeting: same as code inspection

## Code walkthrough...

- ≈ NB: Test cases should be simple, used to drive the process, which finds the bugs through questioning rather than the test cases themselves.
- ≈ Variation: Single-person code walkthrough/code inspection (usually author of code). Experience: Doesn't work.

## Design of test cases

- ≈ **Key question:** Which subgroups of conceivable test cases give highest probability of finding the largest number of bugs?
- ≈ “Obviously” wrong answer: Random testing. (Surprising result: Not that bad, doesn't require that many extra test cases, though must be really *random*.)

## Black-box testing

- ≈ aka *specification-based testing, data-based testing, external testing*
- ≈ Design of test cases on the basis of the program's specification (*what it is do accomplish*)
- ≈ NB: Black-box testing allows design of test cases *before coding* -- key component in *Extreme Programming*.

## White-box testing

- ≈ aka *structural testing, logic-based testing, internal testing*
- ≈ Design of test cases on the basis of the program's text (*how it does what it does*)

## Test design methods

- ≈ Black-box:
  - ≈ equivalence class partitioning,
  - ≈ border value analysis,
  - ≈ cause-effect diagramming,
  - ≈ guessing
- ≈ White-box:
  - ≈ statement coverage
  - ≈ edge coverage
  - ≈ path coverage

## Test strategy

1. If specification contains complex interactions between input conditions, make a cause-effect diagram
2. Perform and classify input partitioning and border case analysis w.r.t. both inputs and outputs;
3. Classify valid and invalid input and output classes.
4. Supplement test cases using guessing.
5. Check test cases with respect to program logic: statement/condition coverage. Supplement systematically for complete/maximal coverage.

## Integration testing

- ⌘ **Big-bang testing:** Put all (tested) modules together and test the resulting system.
- ⌘ **Incremental testing:** Add modules in small chunks and test.
- ⌘ **Bottom-up testing:** Test base modules first, test modules that depend on them after;
- ⌘ **Top-down testing:** Test top modules first (use stubs for base modules).

## High-level testing

- ⌘ **Functionality testing:** Does system satisfy specified functionality?
- ⌘ **System testing:** Does system perform according to intent? (Usability, reliability, etc.)
- ⌘ **Acceptance test:** Is system accepted by its end-users?
- ⌘ **Installation/deployment test:** Bugs/problems in system installation procedure?

## Criteria for termination of testing

- ⌘ **Wrong:** Stop when allocated time has run out.
- ⌘ **Wrong:** Stop when all test cases have run without showing bugs.

## Tool support

- ⌘ Tool support is very important for:
  - ⌘ repeatability of testing,
  - ⌘ management of test suites,
  - ⌘ regression testing (repeated automated execution of test suite),
  - ⌘ integration into development process,
  - ⌘ communication between programmers/testers
- ⌘ **Examples:**
  - ⌘ JUnit
  - ⌘ Application-specific test harnesses and regression test engines

## Proactive testing: Get it right the first time around

- ⌘ Formal verification (nowadays central in critical software)
- ⌘ Design by contract
- ⌘ Defensive programming