Software Testing (Chapter 13)

Mark van den Brand

Technische Universiteit Eindhoven University of Technology

Where innovation starts

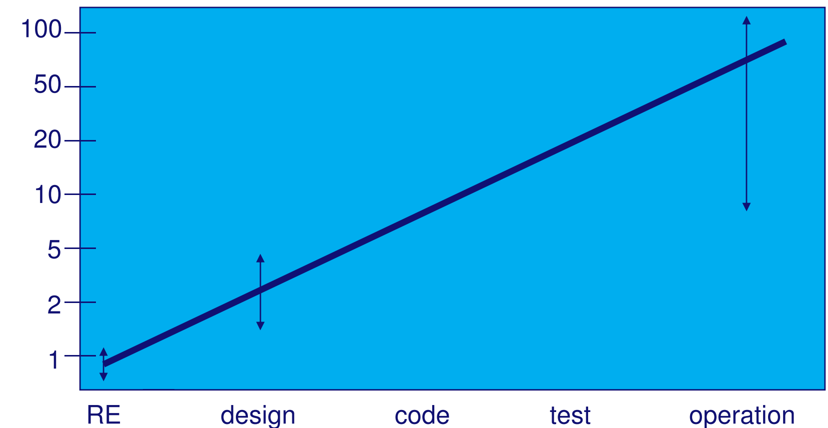
Nasty question

- Suppose you are being asked to lead the team to test the software that controls a new X-ray machine. Would you take that job?
- Would you take it if you could name your own price?
- What if the contract says you'll be charged with murder in case a patient dies because of a mal-functioning of the software?

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



Lessons

- Many errors are made in the early phases
- These errors are discovered late
- Repairing those errors is costly
- ⇒ It pays off to start testing real early

TU/e Technische Universiteit Eindhoven University of Technology

How to proceed?

- Exhaustive testing most often is not feasible
- Random statistical testing does not work either if you want to find errors
- Therefore, we look for systematic ways to proceed during testing

TU/e Technische Universiteit Eindhoven University of Technology

Classification of testing techniques

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

TU/e Technische Universiteit Eindhoven University of Technology

Some preliminary questions

- What exactly is an error?
- How does the testing process look like?
- When is test technique A superior to test technique B?
- What do we want to achieve during testing?
- When to stop testing?

TU/e Technische Universiteit Eindhoven University of Technology

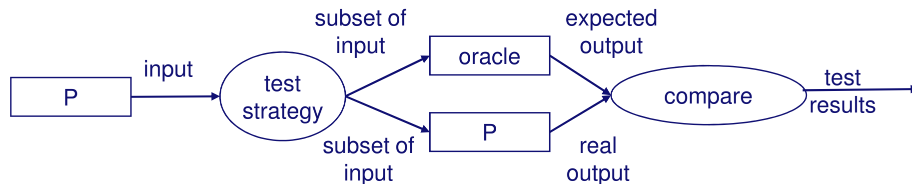
Error, fault, failure

- 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*

When exactly is a failure a failure?

- Failure is a relative notion: e.g. a failure w.r.t. the specification document
- *Verification*: evaluate a product to see whether it satisfies the conditions specified at the start:
  Have we built the system right?
- *Validation*: evaluate a product to see whether it does what we think it should do:
  Have we built the right system?

Testing process



Test adequacy criteria

- Specifies requirements for testing
- Can be used as *stopping rule*: stop testing if 100% of the statements have been tested
- Can be used as *measurement*: a test set that covers 80% of the test cases is better than one which covers 70%
- Can be used as *test case generator*: look for a test which exercises some statements not covered by the tests so far
- A given test adequacy criterion and the associated test technique are opposite sides of the same coin

What is our goal during testing?

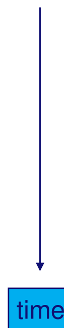
- Objective 1: find as many faults as possible
- Objective 2: make you feel confident that the software works OK

Example constructive approach

- Task: test module that sorts an array A[1..n]. A contains integers; n < 1000
- Solution: take n = 0, 1, 37, 999, 1000. For n = 37, 999, take A as follows:
  - A contains random integers
  - A contains increasing integers
  - A contains decreasing integers
- These are *equivalence classes*: we assume that one element from such a class suffices
- This works if the partition is perfect

Testing models

- *Demonstration*: make sure the software satisfies the specs
- *Destruction*: try to make the software fail
- *Evaluation*: detect faults in early phases
- *Prevention*: prevent faults in early phases



Testing and the life cycle

- requirements engineering
  - criteria: completeness, consistency, feasibility, and testability.
  - typical errors: missing, wrong, and extra information
  - determine testing strategy
  - generate functional test cases
  - test specification, through reviews and the like
- design
  - functional and structural tests can be devised on the basis of the decomposition
  - the design itself can be tested (against the requirements)
  - formal verification techniques
  - the architecture can be evaluated

Testing and the life cycle (cnt'd)

- implementation
  - check consistency implementation and previous documents
  - code-inspection and code-walkthrough
  - all kinds of functional and structural test techniques
  - extensive tool support
  - formal verification techniques
- maintenance
  - regression testing: either retest all, or a more selective retest

TU/e Technische Universiteit Eindhoven University of Technology

Test-Driven Development (TDD)

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

TU/e Technische Universiteit Eindhoven University of Technology

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

TU/e Technische Universiteit Eindhoven University of Technology

Test Stages

- module-unit testing and integration testing
  - bottom-up versus top-down testing
- system testing
- acceptance testing
- installation testing

TU/e Technische Universiteit Eindhoven University of Technology

Manual Test Techniques

- static versus dynamic analysis
- compiler does a lot of static testing
- static test techniques
  - reading, informal versus peer review
  - walkthrough and inspections
  - correctness proofs, e.g., pre-and post-conditions: {P} S {Q}
  - stepwise abstraction

(Fagan) inspection

- Going through the code, statement by statement
- Team with ~4 members, with specific roles:
  - moderator: organization, chairperson
  - code author: silent observer
  - (two) inspectors, readers: paraphrase the code
- Uses checklist of well-known faults
- Result: list of problems encountered

Principles of inspecting

- Inspect the most important documents of all types
  - code, design documents, test plans and requirements
- Choose an effective and efficient inspection team
  - between two and five people
  - Including experienced software engineers
- Require that participants prepare for inspections
  - They should study the documents prior to the meeting and come prepared with a list of defects
- Only inspect documents that are ready
  - Attempting to inspect a very poor document will result in defects being missed

Principles of inspecting

- Avoid discussing how to fix defects
  - Fixing defects can be left to the author
- Avoid discussing style issues
  - Issues like are important, but should be discussed separately
- Do not rush the inspection process
  - A good speed to inspect is
    - 200 lines of code per hour (including comments)
    - or ten pages of text per hour

Principles of inspecting

- Avoid making participants tired
  - It is best not to inspect for more than two hours at a time, or for more than four hours a day
- Keep and use logs of inspections
  - You can also use the logs to track the quality of the design process
- Re-inspect when changes are made
  - You should re-inspect any document or code that is changed more than 20%

Example checklist

- Wrong use of data: variable not initialized, dangling pointer, array index out of bounds, …
- Faults in declarations: undeclared variable, variable declared twice, …
- Faults in computation: division by zero, mixed-type expressions, wrong operator priorities, …
- Faults in relational expressions: incorrect Boolean operator, wrong operator priorities, .
- Faults in control flow: infinite loops, loops that execute n-1 or n+1 times instead of n, ...

Inspecting compared to testing

- Both testing and inspection rely on different aspects of human intelligence.
- Testing can find defects whose consequences are obvious but which are buried in complex code.
- Inspecting can find defects that relate to maintainability or efficiency.
- The chances of mistakes are reduced if both activities are performed.

Testing or inspecting, which comes first?

- It is important to inspect software *before* extensively testing it.
- The reason for this is that inspecting allows you to quickly get rid of many defects.
- If you test first, and inspectors recommend that redesign is needed, the testing work has been wasted.
  - There is a growing consensus that it is most efficient to inspect software befor*e a*ny testing is done.
- Even before developer testing

Coverage-based testing

- Goodness is determined by the coverage of the product by the test set so far: e.g., % of statements or requirements tested
- Often based on control-flow graph of the program
- Three techniques:
  - control-flow coverage
  - data-flow coverage
  - coverage-based testing of requirements

Technische Universiteit Eindhoven University of Technology

Control-flow coverage

- This example is about *All-Nodes coverage*, *statement coverage*
- A stronger criterion: *All-Edges coverage*, *branch coverage*
- Variations exercise all combinations of elementary predicates in a branch condition
- Strongest: *All-Paths coverage* (≡ exhaustive testing)
- Special case: all linear independent paths, the *cyclomatic number criterion*

Technische Universiteit Eindhoven University of Technology

Data-flow coverage

- Looks how variables are treated along paths through the control graph.
- Variables are *defined* when they get a new value.
- A definition in statement X is *alive* in statement Y if there is a path from X to Y in which this variable is not defined anew. Such a path is called *definition-clear*.
- We may now test all definition-clear paths between each definition and each use of that definition and each successor of that node: *All-Uses coverage.*

Technische Universiteit Eindhoven University of Technology

Coverage-based testing of requirements

- Requirements may be represented as graphs, where the nodes represent elementary requirements, and the edges represent relations (like yes/no) between requirements.
- And next we may apply the earlier coverage criteria to this graph

Technische Universiteit Eindhoven University of Technology

Fault-based testing

- In coverage-based testing, we take the structure of the artifact to be tested into account
- In fault-based testing, we do not *directly* consider this artifact
- We just look for a test set with a high ability to detect faults
- Two techniques:
  - Fault seeding
  - Mutation testing

How tests are treated by mutants

- Let P be the original, and P' the mutant
- Suppose we have two tests:
  - T1 is a test, which inserts an element that equals a[k] with k<n
  - T2 is another test, which inserts an element that does not equal an element a[k] with k<n
- Now P and P' will behave the same on T1, while they differ for T2
- In some sense, T2 is a "better" test, since it in a way tests this upper bound of the for-loop, which T1 does not

How to use mutants in testing

- If a test produces different results for one of the mutants, that mutant is said to be *dead*
- If a test set leaves us with many live mutants, that test set is of low quality
- If we have M mutants, and a test set results in D dead mutants, then the *mutation adequacy score* is D/M
- A larger mutation adequacy score means a better test set

Strong vs weak mutation testing

- Suppose we have a program P with a component T
- We have a mutant T' of T
- Since T is part of P, we then also have a mutant P' of P

- In *weak mutation testing*, we require that T and T' produce different results, but P and P' may still produce the same results
- In *strong mutation testing*, we require that P and P' produce different results

Assumptions underlying mutation testing

- *Competent Programmer Hypothesis*: competent programmers write programs that are approximately correct
- *Coupling Effect Hypothesis*: tests that reveal simple faults can also reveal complex faults

Error-based testing

- Decomposes input (such as requirements) in a number of subdomains
- Tests inputs from each of these subdomains, and especially points near and just on the boundaries of these subdomains -- those being the spots where we tend to make errors
- In fact, this is a systematic way of doing what experienced programmers do: test for 0, 1, nil, etc

Comparison of test adequacy criteria

- Criterion A is stronger than criterion B if, for all programs P and all test sets T, X-adequacy implies Y-adequacy
- In that sense, e.g., All-Edges is stronger that All-Nodes coverage (All-Edges “subsumes” All-Nodes)
- One problem: such criteria can only deal with paths that can be executed (are feasible). So, if you have dead code, you can never obtain 100% statement coverage. Sometimes, the subsumes relation only holds for the *feasible* version.

Experimental results

- There is no uniform best test technique
- The use of multiple techniques results in the discovery of *more* faults
- (Fagan) inspections have been found to be very cost effective
- Early attention to testing does pay off