Object-Oriented and Classical Software Engineering

Eighth Edition, WCB/McGraw-Hill, 2011

Stephen R. Schach

CHAPTER 6

TESTING

Overview

Slide 6.3

- Quality issues
- Non-execution-based testing
- Execution-based testing
- What should be tested?
- Testing versus correctness proofs
- Who should perform execution-based testing?
- When testing stops

Testing

Slide 6.4

- There are two basic types of testing
  - Execution-based testing
  - Non-execution-based testing

Testing (contd)

Slide 6.5

- "V & V"
  - *Verification*
    - Determine if the workflow was completed correctly
  - *Validation*
    - Determine if the product as a whole satisfies its requirements

Testing (contd)

Slide 6.6

- Warning
  - The term "verify" is also used for all non-execution-based testing

6.1 Software Quality

Slide 6.7

- Not “excellence”
- The extent to which software satisfies its specifications
- Every software professional is responsible for ensuring that his or her work is correct
  - Quality must be built in from the beginning

6.1.1 Software Quality Assurance

Slide 6.8

- The members of the SQA group must ensure that the developers are doing high-quality work
  - At the end of each workflow
  - When the product is complete
- In addition, quality assurance must be applied to
  - The process itself
    - Example: Standards

6.1.2 Managerial Independence

Slide 6.9

- There must be managerial independence between
  - The development group
  - The SQA group
- Neither group should have power over the other

Managerial Independence (contd)

Slide 6.10

- More senior management must decide whether to
  - Deliver the product on time but with faults, or
  - Test further and deliver the product late
- The decision must take into account the interests of the client and the development organization

6.2 Non-Execution-Based Testing

- Underlying principles
  - We should not review our own work
  - Group synergy

6.2.1 Walkthroughs

- A walkthrough team consists of from four to six members

- It includes representatives of
  - The team responsible for the current workflow
  - The team responsible for the next workflow
  - The SQA group

- The walkthrough is preceded by preparation
  - Lists of items
    - » Items not understood
    - » Items that appear to be incorrect

6.2.2 Managing Walkthroughs

Slide 6.13

- The walkthrough team is chaired by the SQA representative
- In a walkthrough we detect faults, not correct them
  - A correction produced by a committee is likely to be of low quality
  - The cost of a committee correction is too high
  - Not all items flagged are actually incorrect
  - A walkthrough should not last longer than 2 hours
  - There is no time to correct faults as well

Managing Walkthroughs (contd)

Slide 6.14

- A walkthrough must be document-driven, rather than participant-driven
- Verbalization leads to fault finding
- A walkthrough should never be used for performance appraisal

6.2.3 Inspections

- An inspection has five formal steps
  - Overview
  - Preparation, aided by statistics of fault types
  - Inspection
  - Rework
  - Follow-up

Inspections (contd)

- An inspection team has four members
  - Moderator
  - A member of the team performing the current workflow
  - A member of the team performing the next workflow
  - A member of the SQA group
- Special roles are played by the
  - Moderator
  - Reader
  - Recorder

Fault Statistics

Slide 6.17

- Faults are recorded by severity
  - Example:
    - Major or minor
- Faults are recorded by fault type
  - Examples of design faults:
    - Not all specification items have been addressed
    - Actual and formal arguments do not correspond

Fault Statistics (contd)

Slide 6.18

- For a given workflow, we compare current fault rates with those of previous products
- We take action if there are a disproportionate number of faults in an artifact
  - Redesigning from scratch is a good alternative
- We carry forward fault statistics to the next workflow
  - We may not detect all faults of a particular type in the current inspection

Statistics on Inspections

Slide 6.19

- IBM inspections showed up
  - 82% of all detected faults (1976)
  - 70% of all detected faults (1978)
  - 93% of all detected faults (1986)
- Switching system
  - 90% decrease in the cost of detecting faults (1986)
- JPL
  - Four major faults, 14 minor faults per 2 hours (1990)
  - Savings of $25,000 *per inspection*
  - The number of faults decreased exponentially by phase (1992)

Statistics on Inspections (contd)

Slide 6.20

- Warning
- Fault statistics should never be used for performance appraisal
  - "Killing the goose that lays the golden eggs"

6.2.4 Comparison of Inspections and Walkthroughs

Slide 6.21

- Inspection
  - Two-step, informal process
    - » Preparation
    - » Analysis

- Walkthrough
  - Five-step, formal process
    - » Overview
    - » Preparation
    - » Inspection
    - » Rework
    - » Follow-up

6.2.5 Strengths and Weaknesses of Reviews

Slide 6.22

- Reviews can be effective
  - Faults are detected early in the process

- Reviews are less effective if the process is inadequate
  - Large-scale software should consist of smaller, largely independent pieces
  - The documentation of the previous workflows has to be complete and available online

6.2.6 Metrics for Inspections

Slide 6.23

- Inspection rate (e.g., design pages inspected per hour)
- Fault density (e.g., faults per KLOC inspected)
- Fault detection rate (e.g., faults detected per hour)
- Fault detection efficiency (e.g., number of major, minor faults detected per hour)

Metrics for Inspections (contd)

Slide 6.24

- Does a 50% increase in the fault detection rate mean that
  – Quality has decreased? Or
  – The inspection process is more efficient?

6.3 Execution-Based Testing

Slide 6.25

- Organizations spend up to 50% of their software budget on testing
  - But delivered software is frequently unreliable
- Dijkstra (1972)
  - "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"

6.4 What Should Be Tested?

Slide 6.26

- Definition of *execution-based testing*
  - "The process of inferring certain behavioral properties of the product based, in part, on the results of executing the product in a known environment with selected inputs"
- This definition has troubling implications

6.4 What Should Be Tested? (contd)

Slide 6.27

- "Inference"
  – We have a fault report, the source code, and — often — nothing else
- "Known environment"
  – We never can really know our environment
- "Selected inputs"
  – Sometimes we cannot provide the inputs we want
  – Simulation is needed

6.4 What Should Be Tested? (contd)

Slide 6.28

- We need to test correctness (of course), and also
  – Utility
  – Reliability
  – Robustness, and
  – Performance

6.4.1 Utility

Slide 6.29

- The extent to which the product meets the user's needs
  - Examples:
    - Ease of use
    - Useful functions
    - Cost effectiveness

6.4.2 Reliability

Slide 6.30

- A measure of the frequency and criticality of failure
  - Mean time between failures
  - Mean time to repair
  - Time (and cost) to repair the *results* of a failure

6.4.3 Robustness

Slide 6.31

- A function of
  - The range of operating conditions
  - The possibility of unacceptable results with valid input
  - The effect of invalid input

6.4.4 Performance

Slide 6.32

- The extent to which space and time constraints are met

- Real-time software is characterized by *hard* real-time constraints

- If data are lost because the system is too slow
  - There is no way to recover those data

6.4.5 Correctness

Slide 6.33

- A product is correct if it satisfies its specifications

Correctness of specifications

Slide 6.34

- Incorrect specification for a sort:

Input specification:	p : array of n integers, $n > 0$.
Output specification:	q : array of n integers such that $q[0] \leq q[1] \leq \cdots \leq q[n-1]$

Figure 6.1

- Function `trickSort` which satisfies this specification:

```
void trickSort (int p[ ], int q[ ])
{
    int i;
    for (i = 0; i < n; i++)
        q[i] = 0;
}
```

Figure 6.2

Correctness of specifications (contd)

Slide 6.35

- Incorrect specification for a sort:

Input specification:	p : array of n integers, $n > 0$.
Output specification:	q : array of n integers such that $q[0] \le q[1] \le \cdots \le q[n-1]$

Figure 6.1 (again)

- Corrected specification for the sort:

Input specification:	p : array of n integers, $n > 0$.
Output specification:	q : array of n integers such that $q[0] \le q[1] \le \cdots \le q[n-1]$ The elements of array q are a permutation of the elements of array p, which are unchanged.

Figure 6.3

Correctness (contd)

Slide 6.36

- Technically, correctness is
- *Not* necessary
  - Example: C++ compiler
- *Not* sufficient
  - Example: `trickSort`

6.5 Testing versus Correctness Proofs

Slide 6.37

- A correctness proof is an alternative to execution-based testing

6.5.1 Example of a Correctness Proof

Slide 6.38

- The code segment to be proven correct

```
int k, s;
int y[n];
k = 0;
s = 0;
while (k < n)
{
    s = s + y[k];
    k = k + 1;
}
```

Figure 6.4

Example of a Correctness Proof (contd)

Slide 6.39

- A flowchart equivalent of the code segment

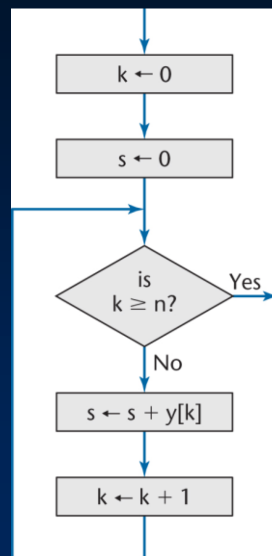

Figure 6.5

Example of a Correctness Proof (contd)

Slide 6.40

- Add
  - Input specification
  - Output specification
  - Loop invariant
  - Assertions

- (See next slide)

Example of a Correctness Proof (contd)

Slide 6.41

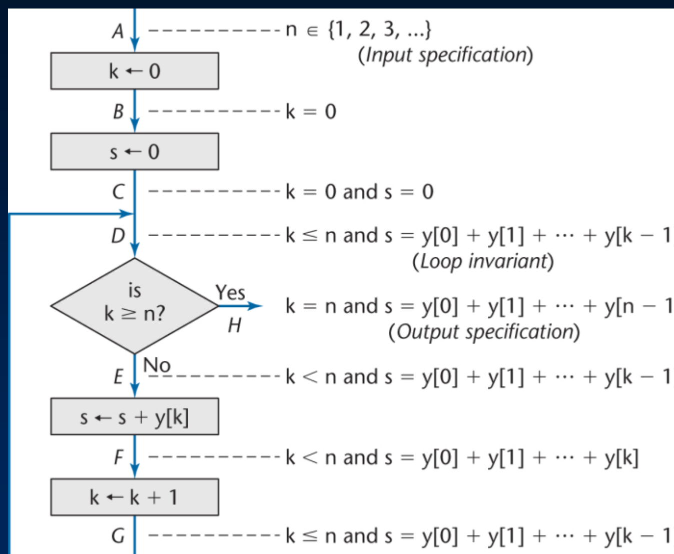

Figure 6.6

Example of a Correctness Proof (contd)

Slide 6.42

- An informal proof (using induction) appears in Section 6.5.1

6.5.2 Correctness Proof Mini Case Study

Slide 6.43

- Dijkstra (1972):
  - "The programmer should let the program proof and program grow hand in hand"
- "Naur text-processing problem" (1969)

Naur Text-Processing Problem

Slide 6.44

- Given a text consisting of words separated by a `blank` or by `newline` characters, convert it to line-by-line form in accordance with the following rules:
- Line breaks must be made only where the given text contains a `blank` or `newline`
- Each line is filled as far as possible, as long as
- No line will contain more than `maxpos` characters

Episode 1

Slide 6.45

- Naur constructed a 25-line procedure
- He informally proved its correctness

Episode 2

Slide 6.46

- 1970 — Reviewer in *Computing Reviews*
  - The first word of the first line is preceded by a `blank` unless the first word is exactly `maxpos` characters long

Episode 3

Slide 6.47

- 1971 — London finds 3 more faults
- Including:
  - The procedure does not terminate unless a word longer than `maxpos` characters is encountered

Episode 4

Slide 6.48

- 1975 — Goodenough and Gerhart find three further faults
- Including:
  - The last word will not be output unless it is followed by a `blank` or `newline`

Correctness Proof Mini Case Study (contd)

- Lesson:

- Even if a product has been proven correct, it must still be tested

6.5.3 Correctness Proofs and Software Engineering

- Three myths of correctness proving (see over)

Three Myths of Correctness Proving

Slide 6.51

- Software engineers do not have enough mathematics for proofs
  - Most computer science majors either know or can learn the mathematics needed for proofs
- Proving is too expensive to be practical
  - Economic viability is determined from cost–benefit analysis
- Proving is too hard
  - Many nontrivial products have been successfully proven
  - Tools like theorem provers can assist us

Difficulties with Correctness Proving

Slide 6.52

- Can we trust a theorem prover ?

```
void theoremProver ( )
{
    print "This product is correct";
}
```

Figure 6.7

Difficulties with Correctness Proving (contd)

Slide 6.53

- How do we find input–output specifications, loop invariants?
- What if the specifications are wrong?
- We can never be sure that specifications or a verification system are correct

Correctness Proofs and Software Engineering (contd)

Slide 6.54

- Correctness proofs are a vital software engineering tool, *where appropriate*:
  - When human lives are at stake
  - When indicated by cost–benefit analysis
  - When the risk of not proving is too great
- Also, informal proofs can improve software quality
  - Use the `assert` statement
- Model checking is a new technology that may eventually take the place of correctness proving (Section 18.11)

6.6 Who Should Perform Execution-Based Testing?

- Programming is *con*structive

- Testing is *de*structive
  – A successful test finds a fault

- So, programmers should not test their own code artifacts

Who Should Perform Execution-Based Testing? (contd)

- Solution:
  – The programmer does informal testing
  – The SQA group then does systematic testing
  – The programmer debugs the module

- All test cases must be
  – Planned beforehand, including the expected output, and
  – Retained afterwards

6.7 When Testing Stops

- Only when the product has been irrevocably discarded