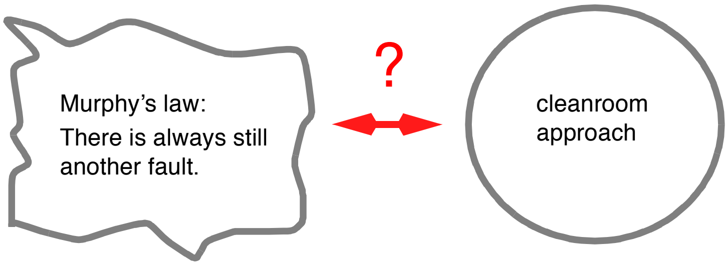
SOFTWARE TESTING - STATE OF THE ART, METHODS, AND LIMITATIONS

MONIKA HEINER

monika.heiner@b-tu.de
http://www.informatik.tu-cottbus.de

PRELIMINARIES



- natural fault rate of seasoned programmers
  -> *about 1-3 % of produced program lines*
- fault-avoidant software **construction** techniques ?
  -> *built-in quality, quality by construction*
- **validation** techniques seem to be unavoidable !

Validation versus Testing

- **VALIDATION**
  -> *any confidence-increasing method to trust in the software's quality*
- undecidability of basic questions in software validation
  - program termination
  - equivalence of programs
  - program verification
  - . . .
- validation == testing
- testing portion of total software production effort
  -> *standard system:* $\geq$ *50 %*
  -> *extreme availability demands:* $\approx$ *80 %*

Correct Software

- A software product is formally correct, if the following three items correspond:
  -> *specification*
  -- the expected properties
  -> *software behavior*
  -- the observed properties
  -> *documentation*
  -- the product description for application and maintenance
- 100% totally correct software is possible !!!
  -> *holds by definition for the empty specification*
- How to **validate** the correspondance ?
  -> *using the software itself*
  -> *using a model of the software instead*
  . . . model-based software validation

TESTING APPROACH

- checking properties
  - -> of the real implementation of the software
  - -> in the real environment

  against its specification / documentation

- by reading it

  -> ***STATIC TESTING (HUMAN TESTING)***

- by executing it

  -> ***DYNAMIC TESTING***

PROPERTIES

- properties

(functional) correctness	robustness, reliability, availability	performance/throughput
safety, security	portability, maintainability, readability	real time behavior/ deadline conformance
usability, stability, ...	extendability, ...	resource consumption, ...

- special properties

  -> *specification, usually checked by dynamic testing*

- general properties

  -> *guidelines, usually checked by static testing*

- testing (as any kind of validation)
  can only be as good as the specifications (guidelines) do be

Two Famous Quotes

- E. W. Dijkstra, 1972:

  "Program testing can be used to show the presence of bugs,
  but never to show their absence !"

- G. J. Myers, 1979:

  "Testing means the execution of a program in order to find bugs."

  -> ***if*** *a test run discovers unknown bugs*
  then *it is called successful*
  else *unsuccessful*
  endif

  -> *testing is an inherently destructive task*

  -> *most programmers are unable to test their own programs*

  -> *ask your favourite enemy to test your programs*

Terminology I

- **BUG** - derivation from expected behavior

  -> *fault*

  -> *error*

  -> *failure*

- **TESTING** - discover the bug

- **DEBUGGING** - fix the bug

- testing ≠ debugging

  -> *done at different times*

  -> *by different people*

  -> *using different techniques*

1.6 Explain the terms: test data, test case, test suite, test oracle.

TERMINOLOGY II

- **TEST DATA** - values for all input data

- **TEST CASE** - complete set of
  values for all input data + corresponding output data values

  -> *A good test case answers one or several questions concerning the test object.*

  -> *Testing is a highly sophisticated task !*

  -> *Test data may be generated, test cases not !*

  The generator would have to have the same function as the software being tested.

- **TEST SUITE** - a representative set of test cases

  -> *table-like test case notation*

- **TEST ORACLE** - assesses a given test case

GENERAL PROCEDURE

- test steps

  (1) derivation of test cases
  (from a suitable system specification)

  -> The outcome is predicted and documented before the test is run !

  (2) execution of these test cases

  (3) assessment of the test results

- what was in the beginning ?

  -> *test object, i. e. software*

  -> *test cases*

- simultaneous design of software and its test cases !

1.5 Why is an exhaustive testing - in the general case - impossible? Give at least five reasons.

TEST CASE SELECTION

- ❑ exhaustive testing impossible
  - all valid inputs -> correctness, . . .

    -> maybe theoretically finite, but mostly practically infinite
  - all invalid inputs -> robustness, security, reliability, . . .

    -> infinite
  - state-preserving software (operating/information systems):
    a (trans-) action depends on its predecessors

    -> all sequences of (trans-) actions had to be regarded !?

- ❑ test case design strategy

  -> *finding good test suites,*

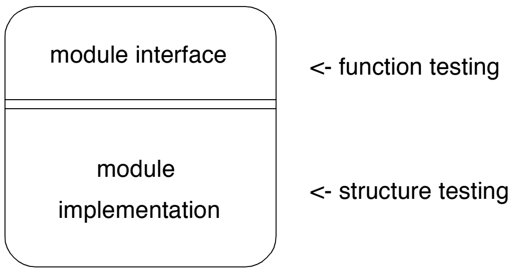
  -> *good = sufficiently small, but high bug discover rate*

BASIC STRATEGIES

- ❑ structure testing (1)

  -> *white-box testing,*
  developer testing

  -> *basis:*
  inner structure of the test object



- ❑ function testing (2)

  -> *black-box testing,*
  user testing

  -> *basis:*
  behavior given by the specification

- ❑ diversified testing (3)

  -> *back-to-back testing, mutation testing, perturbation testing*

(1) Structure Testing

- ❑ based on control structure model (= control flow model)

program elements	control flow graph	Petri net
statements	nodes	transitions
control flow	arcs	places

- ❑ control flow - based testing
- ❑ data flow - based testing (defs/uses methods)
- ❑ **TEST COVERAGE**
  - relation of executed to existing statements/branches/paths . . .
  - easy to compute by code instrumentation
  - side-effect: hot spots are revealed -> tuning
- ❑ main drawback: specification is not checked !

1.7 Give a classification scheme (as a tree) of popular test methods. There should be at least 10 nodes.

(2) Function Testing

- ❑ considerations on the input space
  - -> *equivalence partitioning*
  - -> *boundary value testing*
  - -> *special value testing*

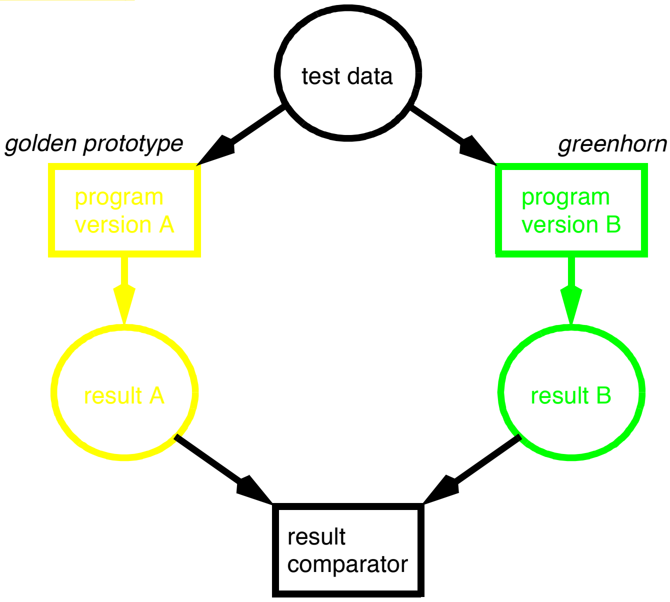
  } *effective selection depends on the skills and experienc of the tester*

- ❑ random testing, statistical testing
  - -> *estimation of residual defects*
  - -> *suitable combination with equivalence partitioning*
- ❑ testing against some model
  - -> *state automaton*
  - -> *cause effect graph*
  - -> *fault tree, . .*

  } test coverages similar to structure testing node/branch/path covrge

(3) Diversified Testing I

- back to back testing



Remark:
Usually, not applicable.

(3) Diversified Testing II

- mutation testing
  - make small changes (mutations) to the program
  - run the mutated program using the same test suite as the program being tested
  - the test suite is adequate, if it finds all mutations

- perturbation testing (fault injection)
  - implementing anomalies for inputs, outputs, and everything in between
  - impact of component bugs on the entire system

    -> fault tolerance

RECOMMENDED PROCEDURE

- ❑ function testing
  - code instrumentation to observe test coverage
  - design test suite using equivalence classes
  - execute test suite neglecting any reached coverage
- ❑ structure testing
  - evaluate reached test coverage
  - design additional test cases to increase test coverage
  - execute additional test cases
  - repeat as long as the specified degree has not been reached
- ❑ mutation test
  - test suite assessment
- ❑ regression testing
  - each debugging requires re-execution of the complete test suite

SUPPORT BY
SUITABLE TEST TOOLS !!

- ❑ Remark:
  Usually, test suites growth step-wise over the time
  by just careful bookkeeping what has been tested before.

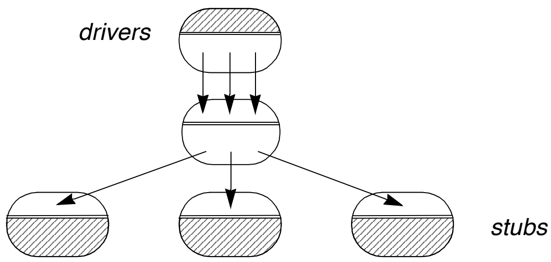
INCREMENTAL TESTING

- ❑ most programs are too complicated to understand all details at a glance
- ❑ white-box testing becomes more and more impractical
  with increasing size of the test component
- ❑ way out: modular programming with sound interfaces (ADT),
  BUT: all interfaces are sources of confusion
- ❑ consequences: step-wise bottom up / top down testing

• unit testing	procedures, . . .
• module testing	set of procedures + interface
• integration testing	interaction of several modules
• system testing	complete software product

MUNCHAUSEN'S TEST MIRACLE

- step-wise testing requires
  - test **DRIVERS**
    simulating the calling modules
  - test **STUBS**
    simulating the called modules



- these test environments must be programmed and tested too,

  . . .

  . . .

  . . . . . .

CLASSIFICATION OF TEST METHODS

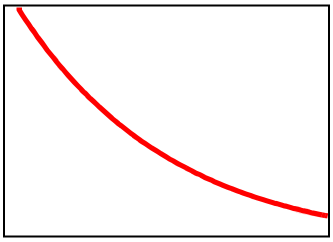
criteria	test method	remarks
kind of test execution	inspection of program code running of executables	review, walk-through, . . .
kind of knowledge of the test object	structure test (white box test, developer test) function test (black box test, user test)	basis: inner structure of the test object basis: behavior given by the specification
size of the test object	unit testing module testing integration testing system testing	procedures, . . . set of procedures + interface interaction of several modules complete software product

SPECIAL CHALLENGES

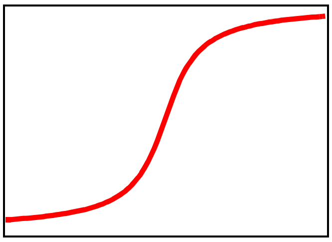
- ❑ testing of alternative programming paradigms using
  - -> *declarative programming languages*
  - -> *functional programming languages*
  - -> *object-oriented programming languages*
- ❑ programs which can hardly be described by an IO function
  - -> *GUI*
  - -> *state-preserving software*
  - -> *reactive systems's software*
- ❑ systematic testing of concurrent programs
  - -> *is much more complicated than of sequential ones*

CRITERIA TO FINISH TESTING

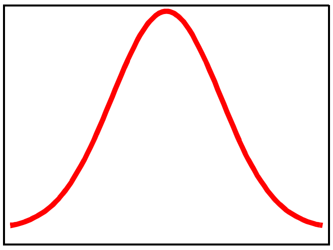
- ❑ common
  - time is over (time-to-market pressure)
  - all test cases successful
- ❑ better (?)
  - Discover a given amount of bugs !
  - Reach a specified degree of test coverage(s) !
  - Reach a specified fault rate ! (number of found bugs per time)



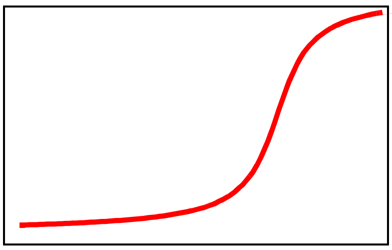
optimistic view



pessimistic view



realistic view (?)



ageing model

STATE OF THE ART

- ❑ effective testing is still a challenge in real-life software development
- ❑ validation needs knowledgeable professionals
  - -> *study / job specialization*
  - -> *profession of "software tester"*
- ❑ testing is very time and resource consuming
  - -> *'external' quality pressure*
- ❑ There is no such thing as a fault-free program !
  - -> *sufficient dependability for a given user profile*
  - -> *how to characterize a user profile ?*
- ❑ sophisticated testing is not manageable without tool support ->exercises

LIMITATIONS OF TESTING

- ❑ Testing (as any kind of validation) is no substitute for thinking !
- ❑ testing can only be as good as the specification
  - -> *readable* <-> *unambiguous*
  - -> *complete* <-> *limited size*
- ❑ (dynamic) testing needs an executable
- ❑ "Program testing can be used to show the presence of bugs, but never to show their absence !" [Dijkstra 72]
  - sophisticated *static analyses (***CONTEXT CHECKING***)* to prove the absence of certain types of bugs
  - *correctness proofs (***VERIFICATION***),* similar to the proof of a mathematical theorem

next slide

Software Validation Methods