TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Department of computer Engineering
ati.ttu.ee

IAF0530/IAF9530

# Süsteemide usaldusväärsus ja veakindlus
# Dependability and fault tolerance

Loeng 4
Fault Tolerance, Software Testing

**Gert Jervan**
gert.jervan@pld.ttu.ee

Tallinn University of Technology
Department of Computer Engineering
Estonia



## Important

- ✓ February 28: Case study topic selection, incl. preliminary list of literature (by e-mail)
- ✓ No lecture on February 28



TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Department of computer Engineering
ati.ttu.ee

## Fault Tolerance



## Basics

- ✓ Computing systems are characterized by five fundamental properties:
  - functionality
  - usability
  - performance
  - cost
  - dependability





## Faults

- ✓ Faults are there!
- ✓ Either prevent, **tolerate**, remove or forecast
- ✓ We need redundancy
  - System that is more complex than needed for performing the required task





## Means to Achieve Dependability

- ✓ Fault prevention
  - Good design processes, avoid design flaws
  - Good procedures for runtime faults
- ✓ Fault tolerance
  - Fault detection
  - Redundancy
  - Diversity
- ✓ Fault removal
  - Verification and validation during design
  - Corrective/preventive action during maintenance
- ✓ Fault forecasting
  - Simulation, modelling, prediction
  - Analysis based on history statistics

## Fault Tolerance

- Automobile:
  - Spare Tires
  - Dual Braking Systems
- Power Supplies:
  - UPS/battery backup
  - Power-fail interrupts
- Multiple engines on aircraft
- Emergency lighting in buildings
- Tape backups of disk files
- Checkpoint/restart of long-running programs
- Parity and SECDED in computer memories





## Faults

- Random faults (Degradation faults)
  - Arise during operation
  - Usually hardware component failure
- Systematic faults (Design Faults)
  - mistakes in the spec
  - mistakes in the hardware
  - mistakes in the software





## Faults

- Faults are either permanent, transient or intermittent
- Design faults are always permanent
- Dealing with faults:
  - During development: fault avoidance & removal
  - During operation: fault tolerance & detection





## Hardware Faults

- Use of fault models
- Decomposition into modules
  - Gates, transistors, etc
- Connection faults
  - Single stuck-at model, bridging model (shorts), stuck-open
- Used to model hardware faults
  - Design testing schemes for digital circuits
  - Fault removal coverage usually less than 100%
  - Guard against physical defects, not design faults
- In safety critical systems
  - Combined with Failure Modes and Effects Analysis (FMEA)
  - Need fault avoidance by verification…





## Other Faults

- Hardware design and specification faults
  - Few fault models available
  - Many faults cannot be modelled
  - System must meet the spec, but spec might be incorrect as well
  - Spec errors may manifest as either hardware or software failures
  - Use of formal methods (formal spec. languages, automata theory, formal verification, model checking, etc.)





## Software Faults

- Bugs:
  - Software spec faults
  - Coding faults
  - Logical errors within calculations
  - Stack overflows or underflows
  - Uninitialized variables
- No random failures and it does not degrade with age
- Always systematic
- Exhaustive testing almost impossible
- Must be tolerated

## SW Testing - i.e. Verification

- Verification:
  - SW testing
  - formal verification
- Functional and structural testing
- Path testing, transaction flow testing, data-flow testing, domain testing, mutation testing etc.





## Fault Detection Techniques

- Functionality checking
  - march test
- Consistency checking
  - range checking, overflow
- Signal comparison
- Information redundancy
  - checksums, cyclic redundancy codes, error correcting codes
- Monitoring techniques
  - Loopback testing
  - Power supply monitoring





## Watchdog Timer

- An inexpensive method of error detection
- Process being watched must reset the timer before the timer expires, otherwise the watched process is assumed as faulty
- Watchdog timers only detect errors which manifest themselves as a control-flow error such that the system does not continue to reset the timer
- Only processes with relatively deterministic runtimes can be checked, since the error detection is based entirely on the time between timer resets





## Heartbeats

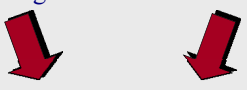
- A common approach to detecting process and node failures in a distributed (networked) computing environment.
- Periodically, a monitoring entity sends a message (a heartbeat) to a monitored node or process and waits for a reply.
- If the monitored node does not respond within a predefined timeout interval, the node is declared as failed and appropriate recovery action is initiated.
- Adaptive or smart





## System Testing

HW Testing SW Testing



HW/SW Testing
(system testing)



TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Department of computer Engineering
ati.ttu.ee

**Software Testing**

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Department of computer Engineering
ati.ttu.ee

**Programmers are in a race with the Universe to create bigger and better idiot-proof programs.**

**While the Universe is trying to create bigger and better idiots.**

**So far the Universe is winning**



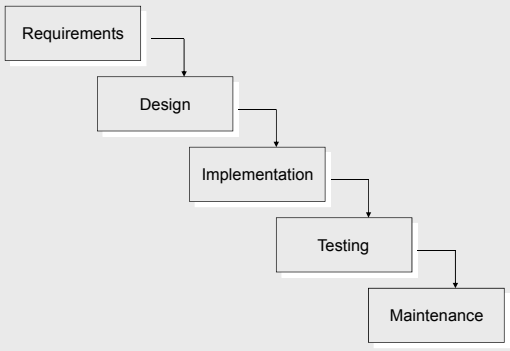
## Software Testing Topics

- ✓ Test Economics
- ✓ Types of Testing
- ✓ Testing coverage





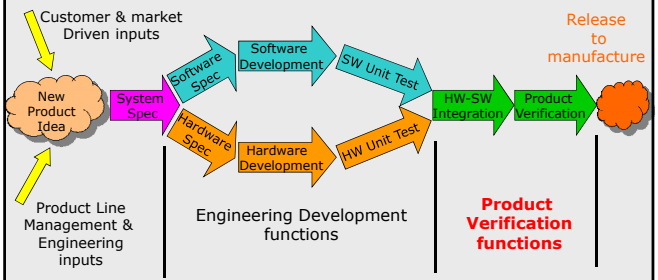
## Software Life Cycle







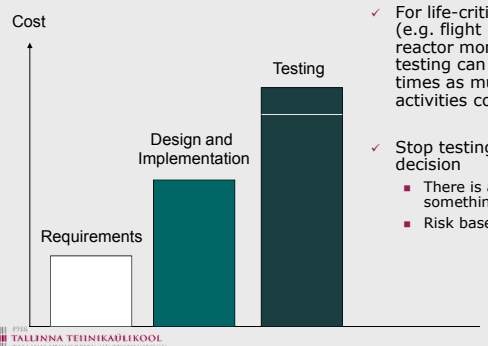

## The Product Development Cycle








## Software Development Costs

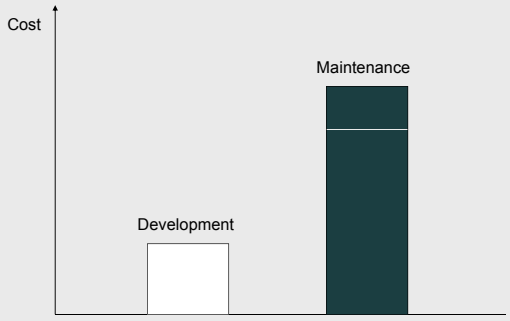



- ✓ For life-critical software (e.g. flight control, reactor monitoring), testing can cost 3 to 5 times as much as all other activities combined.
- ✓ Stop testing is a business decision
  - There is always something more to test
  - Risk based decision





## Software Life Cycle Costs



## Software Qualities

- ✓ Correctness
- ✓ Reliability (dependability)
- ✓ Robustness
- ✓ Safety
- ✓ Security (survivability)
- ✓ Performance
- ✓ Productivity
- ✓ Maintainability, portability, interoperability, …





## Software Verification and Validation

- ✓ Verification
  - Are we building the product right?
  - Process-oriented
    - Does the product of a given phase fulfill the requirements established during the previous phase?
- ✓ Validation
  - Are we building the right product?
  - Product-oriented
    - Does the product of a given phase fulfill the user's requirements?





## Techniques for V&V

- ✓ Static
  - Collects information about a software without executing it
    - Reviews, walkthroughs, and inspections
    - Static analysis
    - Formal verification
- ✓ Dynamic
  - Collects information about a software with executing it
    - Testing: finding errors
    - Debugging: removing errors





## Static Analysis

- ✓ Control flow analysis and data flow analysis
  - Extensively used for compiler optimization and software engineering
- ✓ Examples
  - Unreachable statements
  - Variables used before initialization
  - Variables declared but never used
  - Variables assigned twice but never used between assignments
  - Variables used twice with no intervening assignment
  - Possible array bound violations





## Formal Verification

- ✓ Given a model of a program and a property, determine whether the model satisfies the property based on mathematics
- ✓ Examples
  - Safety
    - If the light for east-west is green, then the light for south-north should be red
  - Liveness
    - If a request occurs, there should be a response eventually in the future

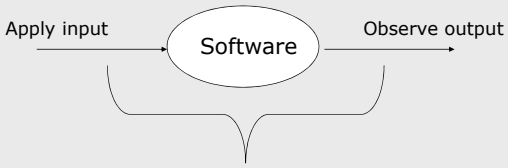




## Introduction to Testing

- ✓ Debugging and testing are not the same thing!
- ✓ Testing is a systematic attempt to break a program.
  - Correct, bug-free programs by construction are the goal but until that is possible (if ever!) we have testing.
  - Since testing is basically **destructive** in nature, it requires that the tester discard *preconceived* notions of the *correctness* of the software to be tested

## Testing




Validate the observed output

Is the observed output the same as the expected output?





## Software Testing Fundamentals

- Testing objectives include
  - Testing is a process of executing a program with the intent of finding an error.
  - A **good** test case is one that has a high probability of finding an as yet undiscovered error.
  - A **successful** test is one that uncovers an as yet undiscovered error.





## Limitations of Testing (I)

- To test all possible inputs is impractical or impossible

```
int foo(int x) {
        y = very-complex-computation(x);
        write(y);
}
```

- To test all possible paths is impractical or impossible

```
int foo(int x) {
        for (index = 1; index < 10000; index++)
                write(x);
}
```





## Limitations of Testing (II)

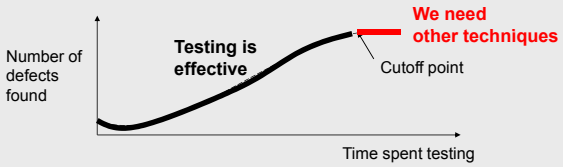
- Dijkstra, 1972
  - Testing can be used to show the presence of bugs, but never their absence
- Goodenough and Gerhart, 1975
  - Testing is successful if the program fails
- The (modest) goal of testing
  - Testing cannot guarantee the correctness of software but can be effectively used to find errors (of certain types)





## Economics of Testing (I)

- The characteristic S-curve for error removal

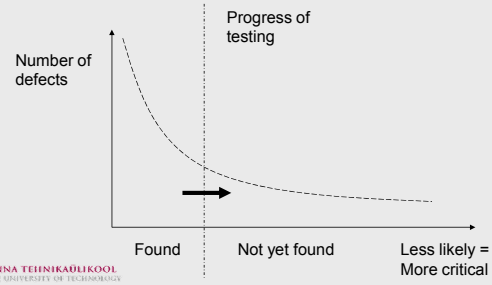






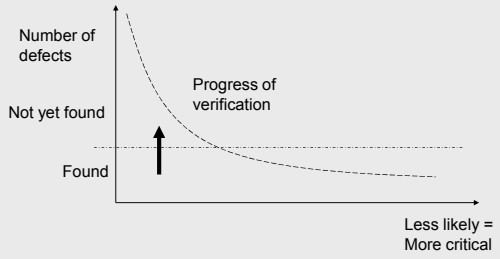

## Economics of Testing (II)

- Testing tends to intercept errors in order of their probability of occurrence



## Economics of Testing (III)

- Verification is insensitive to the probability of occurrence of errors







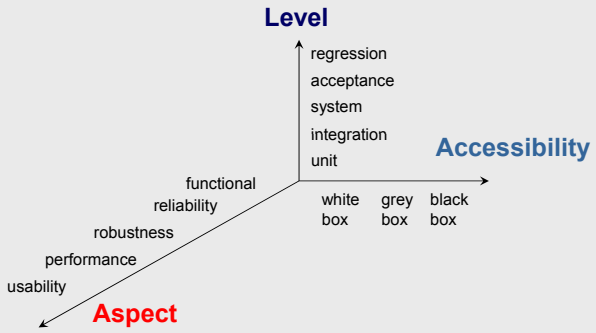

## Fundamental Questions in Testing

- When can we stop testing?
  - Test coverage
- What should we test?
  - Test generation
- Is the observed output correct?
  - Test oracle
- How well did we do?
  - Test efficiency
- Who should test your program?
  - Independent V&V





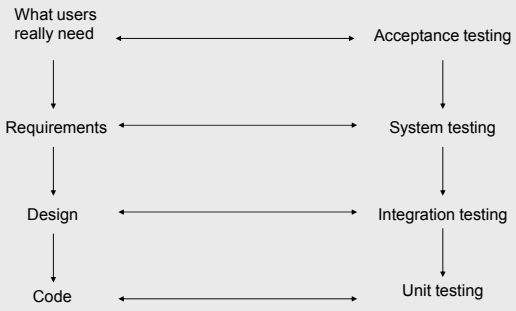
## Types of Testing








## Levels of Testing








## Component/Unit Testing (I)

- A unit of testing
  - Functions in procedural programming languages such as C, Fortran, …

```
Test driver     F1(int x1, y1) {
                    ……
                    F2(x1+1, y1-1);
                }

Test unit       F2(int x2, y2) {
                    ……
                    F3(x2+2, y2-1);
                }

Test stub       F3(int x3, y3) {
                    ……
                }
```





## Component/Unit Testing (II)

- Require knowledge of code
  - High level of detail
  - Deliver thoroughly tested components to integration
- Stopping criteria
  - Code Coverage
  - Quality

## Component/Unit Testing (III)

- Test case
  - Input, expected outcome, purpose
  - Selected according to a strategy, e.g., branch coverage
- Outcome
  - Pass/fail result
  - Log, i.e., chronological list of events from execution





## Integration Testing (I)

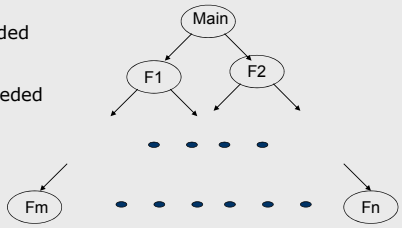
- Interactions among units (assembled components that must be tested and accepted previously)
  - Import/export type compatibility
  - Import/export range errors
    - F1 calls F2 with a parameter of array
    - F1 assumes array of size 8, while F2 assumes an array of size 10
  - Import/export representation
    - F1 calls F2 with a parameter Elapsed_time
    - F1 thinks in seconds, while F2 thinks in miliseconds





## Integration Testing (II)

- Strategies for integration testing
  - Top-down
    - Stubs are needed
  - Bottom-up
    - Drivers are needed
  - Big-bang
  - Functional
  - Drivers & stubs have to tested as well!








## System Testing (I)

- Tests the overall system (the integrated hardware and software) to determine whether the system meets its requirements
- Focuses on the use and interaction of system functionalities rather than details of implementations
- Test cases derived from specification
- Should be carried out by a group independent of the code developers
- Should be planned with the same rigor as other phases of the software development
- Use-case focus





## System Testing (II)

- Non-functional testing
- Quality attributes
  - Performance, can the system handle required throughput?
  - Reliability, obtain confidence that system is reliable
  - Timeliness, testing whether the individual tasks meet their specified deadlines
  - etc.





## Acceptance Testing

- User (or customer) involved
- Environment as close to field use as possible
- Focus on:
  - Building confidence
  - Compliance with defined acceptance criteria in the contract

## Re-Test and Regression Testing (I)

- Conducted after a change
- Re-test aims to verify whether a fault is removed
  - Re-run the test that revealed the fault
- Regression test aims to verify whether new faults are introduced
  - Re-run all tests
  - Should preferably be automated





## Re-test & Regression Testing (II)

- Development versus maintenance
  - Development costs: 1/3
  - Maintenance costs: 2/3
- Testing in maintenance phase
  - How can we test modified or newly inserted programs?
    - Ignore old test suites and make new ones from the scratch or
    - Reuse old test suites and reduce the number of new test suites as many as possible





## Accessibility of Testing

- White box testing (structural testing, program-based testing)
- White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Test cases can be derived that
  - guarantee that all independent paths within a module have been exercised at least once,
  - exercise all logical decisions on their true and false sides,
  - execute all loops at their boundaries and within their operational bounds, and
  - exercise internal data structures to ensure their validity.





## Accessibility of Testing (II)

- Black box testing (functional testing, specification-based testing)
  - Assumes that the program is unavailable or testers do not want to look at the details of the program
    - Derives test cases from the requirements of the program
    - Controls and observes the program only through external interfaces
    - Ideally done by independent test group (not original programmer)
- Grey box testing





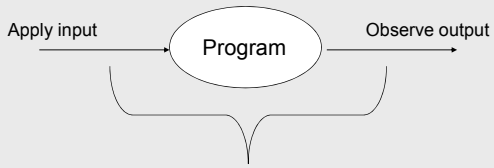
## Program-Based Testing (I)

- Main steps
  - Examine the internal structure of a program
  - Design a set of inputs satisfying a coverage criterion
  - Apply the inputs to the program and collect the actual outputs
  - Compare the actual outputs with the expected outputs
- Limitations
  - Cannot catch omission errors
    - What requirements are missing in the program?
  - Cannot provide test oracles
    - What is the expected output for an input?





## Program-Based Testing (II)




Validate the observed output against the expected output

Who will take care of test oracles?

## Statement Coverage

- Statement coverage of a set of test cases is defined to be the proportion of statements in a unit covered by those test cases.
- 100% statement coverage for a set of tests means that all statements are covered by the tests. That is, all statements will be executed at least once by running the tests.





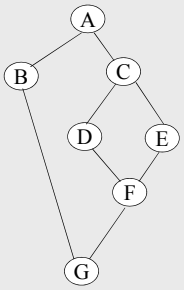
## Branch Coverage

- Branch coverage is determined by the proportion of decision branches that are exercised by a set of proposed test cases.
- 100% branch coverage is where every decision branch in a unit is visited by at least one test in the set of proposed test cases.





## Example – Branch coverage

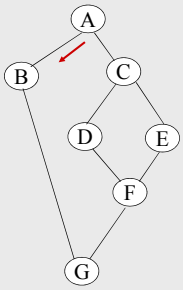



**What branch coverage is achieved by ABG, ACDFG, ACEFG?**





## Example – Branch coverage

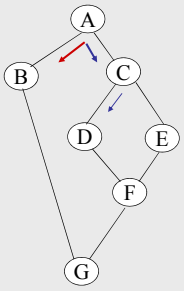



**What branch coverage is achieved by ABG, ACDFG, ACEFG?**





## Example – Branch coverage

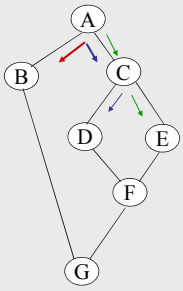



**What branch coverage is achieved by ABG, ACDFG, ACEFG?**



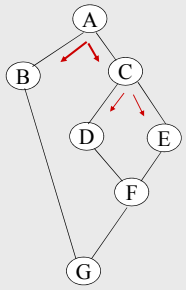


## Example – Branch coverage




**What branch coverage is achieved by ABG, ACDFG, ACEFG?**

## Example – Branch coverage




**What branch coverage is achieved by ABG, ACDFG, ACEFG?**

**4 in total.**

**4 covered**

**So 4/4 = 100% branch coverage**





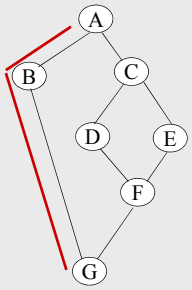
## Path Coverage

- Path coverage is determined by assessing the proportion of execution paths through a unit exercised by the set of proposed test cases.
- 100% path coverage is where every path in the unit is executed at least once by the set of proposed test cases.
- 100% path coverage is achieved by an ideal test set. As we saw the other week, it is all but impossible or infeasible in most programs of any size.





## Example – Path coverage

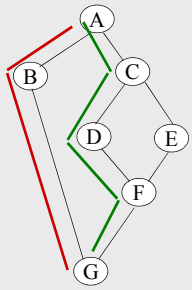



**What path coverage is achieved by ABG, ACDFG, ACEFG?**





## Example – Path coverage

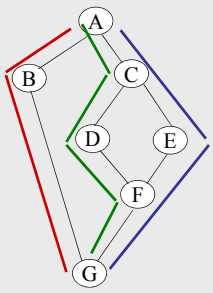



**What path coverage is achieved by ABG, ACDFG, ACEFG?**





## Example – Path coverage




**What path coverage is achieved by ABG, ACDFG, ACEFG?**

**3/3=100%**

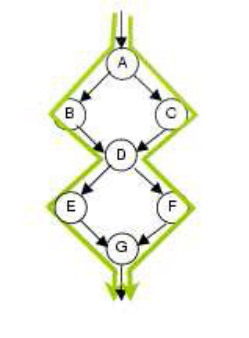




## Coverage

- It is possible to have 100% statement coverage without 100% branch coverage
- It is possible to have 100% branch coverage without 100% path coverage
- 100% path coverage implies 100% branch coverage and 100% branch coverage implies 100% statement coverage

## An example

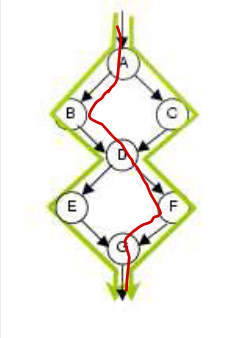



- Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- They, however, only cover 2/4 paths (50%).





## An example

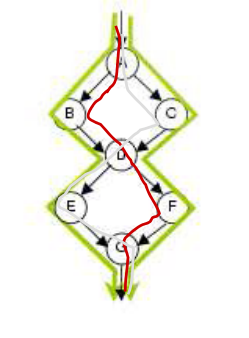



- Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- They, however, only cover 2/4 paths (50%).
- 2 more tests are required to achieve 100% path coverage
  - ABDFG





## An example




- Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- They, however, only cover 2/4 paths (50%).
- 2 more tests are required to achieve 100% path coverage
  - ABDFG, ACDEG





## Loop Testing

- It is usually impossible or infeasible to test all paths in a program involving loops
- Basis Path Testing
  - Zero path: Test zero iterations of the loop body (Guard is negated by loop initialisation)
  - One path: Test a single iteration of the loop body (Good idea to try for 100% path coverage of loop body if loop body is not iterative)
  - Does not consider maximum iteration termination in many cases
  - Does not consider combinations of loop body paths in successive iterations





## Mutation testing

- Create a number of mutants, i.e., faulty versions of program
  - Each mutant contains one fault
  - Fault created by using mutant operators
- Run test on the mutants (random or selected)
  - When a test case reveals a fault, save test case and remove mutant from the set, i.e., it is killed
  - Continue until all mutants are killed
- Results in a set of test cases with high quality
- Need for automation

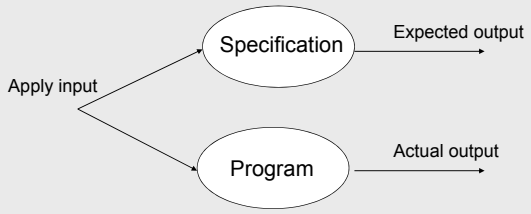




## Specification-Based Testing (I)

- Main steps
  - Examine the structure of the program's specification
  - Design a set of inputs from the specification satisfying a coverage criterion
  - Apply the inputs to the specification and collect the expected outputs
  - Apply the inputs to the program and collect the actual outputs
  - Compare the actual outputs with the expected outputs
- Limitations
  - Specifications are not usually available
    - Many companies still have only code, there is no other document.

## Specification-Based Testing (II)




Validate the observed output against the expected output





## Steps to Testing Nirvana

- Think about potential problems as you design and implement. Make a note of them and develop tests that will exercise these problem areas.
  - Document all loops and their boundary conditions, all arrays and their boundary conditions, all variables and their range of permissible values.
  - Pay special attention to parameters from the command line and into functions and what are their valid and invalid values.
  - Enumerate the possible combinations and situations for a piece of code and design tests for all of them.
  - GIGO - what happens when garbage goes in?

*Kernighan, Pike, "The Practice of Programming"*





## Steps to Testing Nirvana

- Test systematically, starting with easy tests and working up to more elaborate ones.
  - Often leads to "bottom up" testing, starting with simplest modules at the lowest level of calling
  - When those are working, test their callers
  - Document (and/or automate) this testing so that it can be repeated (regression testing) constantly as the code grows and changes.





## Steps to Testing Nirvana

- Within a module, test *incrementally* as you code
  - Write, test, add more code, test again, repeat
  - The earlier that errors are detected, the easier they are to locate and fix.
  - Testing is not only concerning code
    - Documents and models should also be subject to testing





## Tricks of the Trade

- Test boundary conditions.
  - loops and conditional statements should be checked to ensure that loops are executed the correct number of times and that branching is correct
  - if code is going to fail, it usually fails at a boundary
  - check for off-by-one errors, empty input, empty output

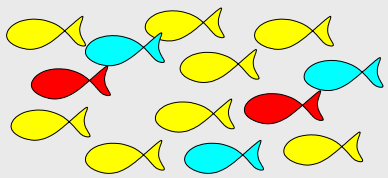




## The Budget Coverage Criterion

- A common answer to "when is testing done"
  - When the money is used up
  - When the deadline is reached
- This is sometimes a rational approach!
  - Implication 1: Test selection is more important than stopping criteria per se.
  - Implication 2: Practical comparison of approaches must consider the cost of test case selection

## Test Selection vs. Test Adequacy



Mutation Testing Example

- Red fish = real program faults (unknown population)
- Blue fish = seeded faults (e.g., mutations) or representative behaviors (known population)
- Adequacy: count blue fish caught, estimate red fish
- Misuse for selection: use special bait to catch blue fish





## Test Selection: Standard Advice

- Specification coverage is good for selection as well as adequacy
  - applicable to informal as well as formal specs
- + Fault-based tests
  - usually ad hoc, sometimes from check-lists
- Program coverage last
  - to suggest uncovered cases, not just to achieve a coverage criterion





## The Importance of Oracles

- Much testing research has concentrated on adequacy, and ignored oracles
- Much testing practice has relied on the "eyeball oracle"
  - Expensive, especially for regression testing
    - makes large numbers of tests infeasible
  - Not dependable
- Automated oracles are essential to cost-effective testing





## Sources of Oracles

- Specifications
  - sufficiently formal (e.g., SCR tables)
  - but possibly incomplete (e.g., assertions in Anna, ADL, APP, Nana)
- Design, models
  - treated as specifications, as in protocol conformance testing
- Prior runs (capture/replay)
  - especially important for regression testing and GUIs; hard problem is parameterization





## What can be automated?

- Oracles
  - assertions; replay; from some specifications
- Selection (Generation)
  - scripting; specification-driven; replay variations
  - selective regression test
- Coverage
  - statement, branch, dependence
- Management





## Design for Test: Principles

Adapted from circuit and chip design

- Observability
  - Providing the right interfaces to observe the behavior of an individual unit or subsystem
- Controllability
  - Providing interfaces to force behaviors of interest
- Partitioning
  - Separating control and observation of one component from details of others

**Remarks by Bill Gates**
17th Annual ACM Conference on Object-Oriented Programming, Seattle, Washington, November 8, 2002

- "… When you look at a big commercial software company like Microsoft, there's actually as much testing that goes in as development. We have as many testers as we have developers. Testers basically test all the time, and developers basically are involved in the testing process about half the time…
- … We've probably changed the industry we're in. We're not in the software industry; we're in the testing industry, and writing the software is the thing that keeps us busy doing all that testing."

1918 TALLINNA TEHNIKAÜLIKOOL TALLINN UNIVERSITY OF TECHNOLOGY
Department of computer Engineering ati.ttu.ee

**Remarks by Bill Gates (cont.)**

- "…The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one."
- "… Well, one of the interesting questions is, when you change a program, … what portion of these test cases do you need to run?"