IAF0030
Arvutitehnika erikursus I

Software Testing

Gert Jervan

Arvutitehnika instituut (ATI)
Tallinna Tehnikaülikool

Tallinn University of Technology

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

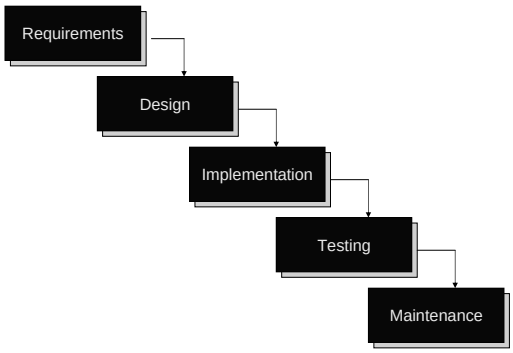
While the Universe is trying to create bigger and better idiots.

So far the Universe is winning

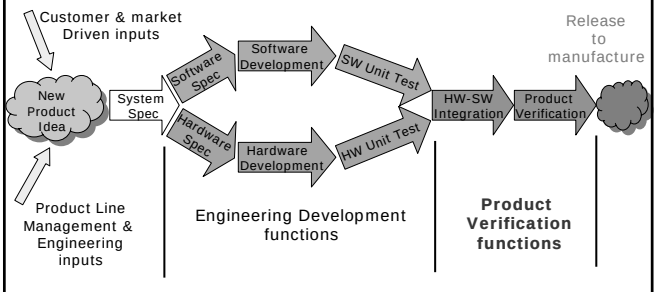
Lecture Outline

- **Introduction**
- **Test Economics**
- **Types of Testing**
- **Testing coverage**

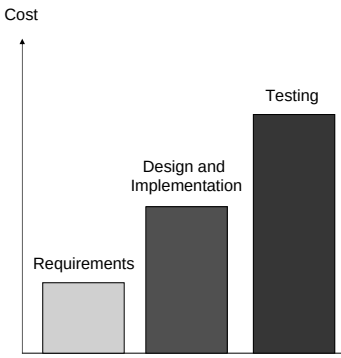
Software Life Cycle



The Product Development Cycle

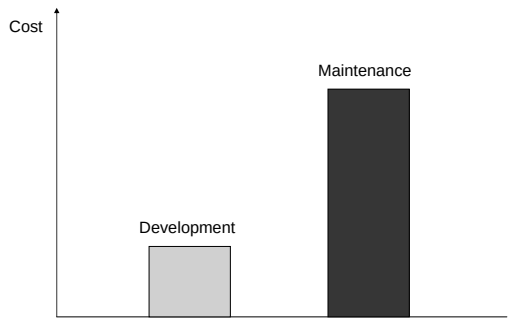


Software Development Costs



- 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

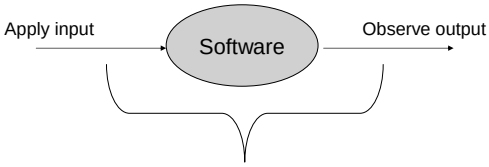
Formal Verification

- 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

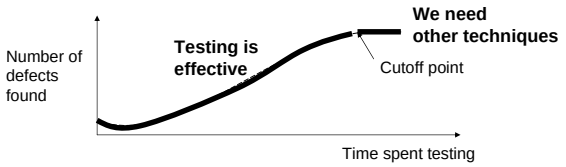
```
int foo(int x) {
    for (index = 1; index < 10000; index++)
        write(x);
}
```


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)

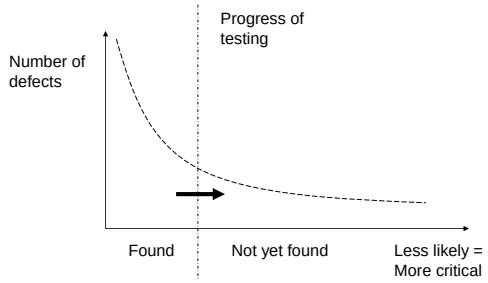
Economics of Testing (I)

- The characteristic S-curve for error removal



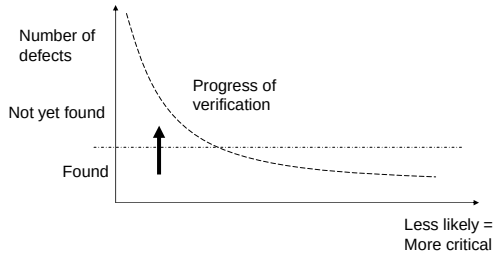
Economics of Testing (II)

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



Economics of Testing (III)

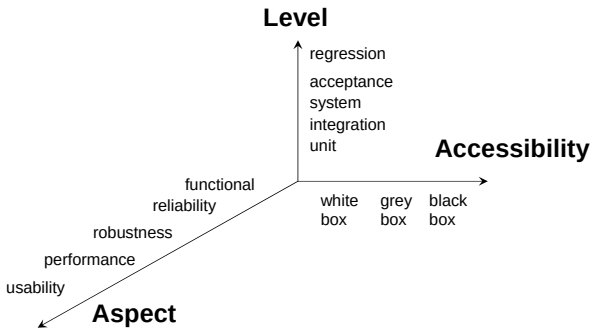
- Verification is insensitive to the probability of occurrence of errors



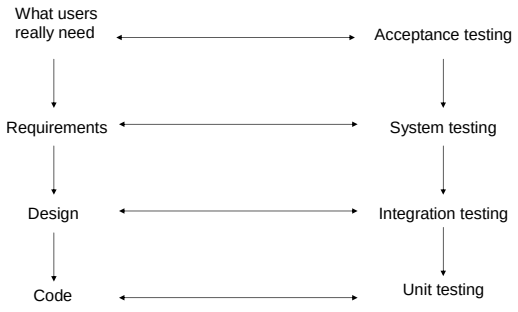
Fundamental Questions in Testing

- 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)

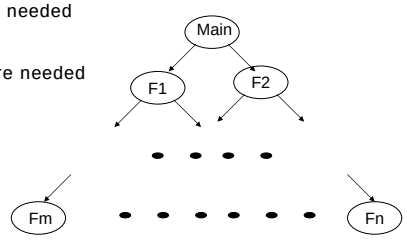
- Test case
  - Input, expected outcome, purpose
  - Selected according to a strategy, e.g., branch coverage
- Outcome
  - Pass/fail result
  - Log, i.e., chronological list of events from execution

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.

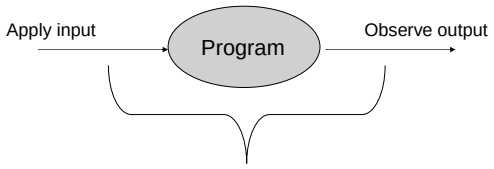
Accessibility of Testing (II)

- 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?

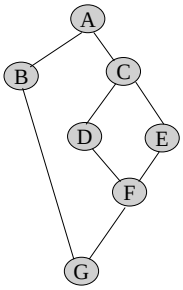
Statement Coverage

- 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

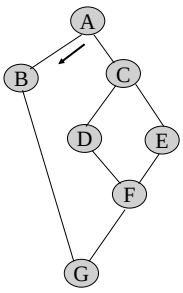
- Branch coverage is determined by the proportion of decision branches that are exercised by a set of proposed test cases.
- 100% branch coverage is where every decision branch in a unit is visited by at least one test in the set of proposed test cases.

Example – Branch coverage



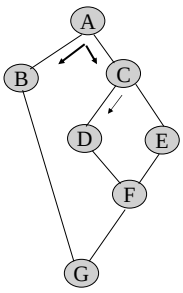
What branch coverage is achieved by ABG, ACDFG, ACEFG?

Example – Branch coverage



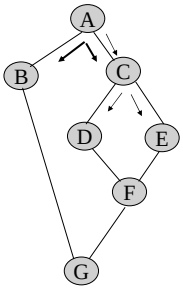
What branch coverage is achieved by ABG, ACDFG, ACEFG?

Example – Branch coverage



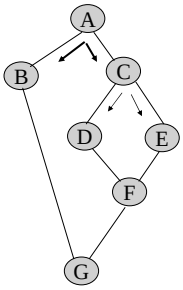
What branch coverage is achieved by ABG, ACDFG, ACEFG?

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.

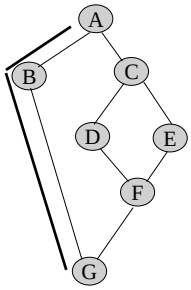
4 covered

So 4/4 = 100% branch coverage

Path Coverage

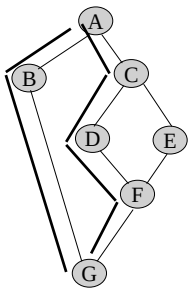
- Path coverage is determined by assessing the proportion of execution paths through a unit exercised by the set of proposed test cases.
- 100% path coverage is where every path in the unit is executed at least once by the set of proposed test cases.
- 100% path coverage is achieved by an ideal test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

Example – Path coverage



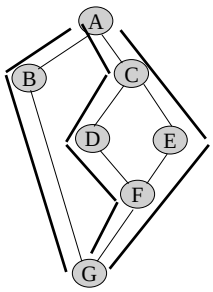
What path coverage is achieved by ABG, ACDFG, ACEFG?

Example – Path coverage



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

Example – Path coverage



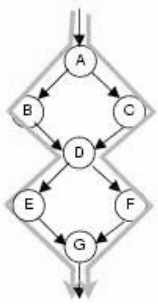
What path coverage is achieved by ABG, ACDFG, ACEFG?

3/3=100%

Coverage

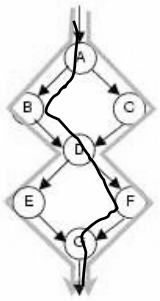
- It is possible to have 100% statement coverage without 100% branch coverage
- It is possible to have 100% branch coverage without 100% path coverage
- 100% path coverage implies 100% branch coverage and 100% branch coverage implies 100% statement coverage

An example



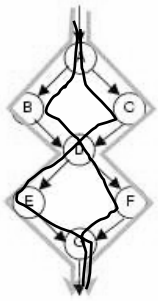
- Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- They, however, only cover 2/4 paths (50%).

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

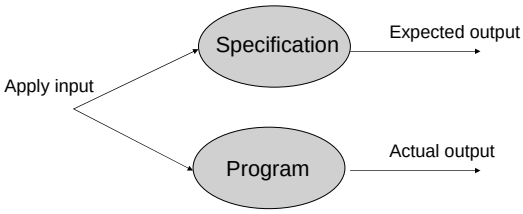
Mutation testing

- 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

Object-Oriented Program Testing

- Unit testing for OO Programs
  - A class is a set of variables and member functions
  - 50% of member functions are just 10 lines of code
  - A class is often a unit of testing in C++ or Java
- Integration testing for OO Programs
  - Rule of thumb in OO development
    - Make a large number of small classes in a bottom-up fashion
  - There are several relationships between classes
    - Association, aggregation, inheritance, concurrency

Steps to Testing Nirvana

Kernighan, Pike, "The Practice of Programming"

- 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?

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

Questions?

Gert Jervan

Department Of Computer Engineering
Tallinn University of Technology
Estonia