Software Testing

Adapted from FSE'98 Tutorial by Michal Young and Mauro Pezze'

Why testing and analysis

- Software is never correct no matter what developing testing technique is used
- All software must be verified
- Software testing is
  - important to control the quality of the product and the process
  - very (often too) expensive

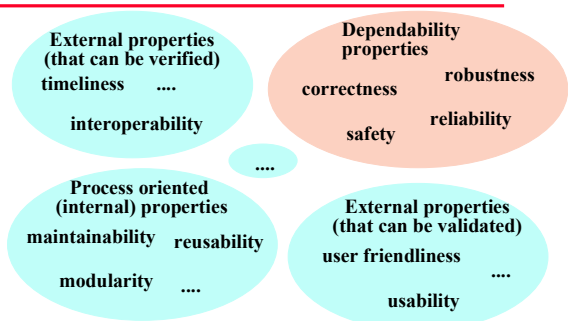
Outline

- A framework for testing and analysis
- Overview of software testing
- Unit testing
- Summary

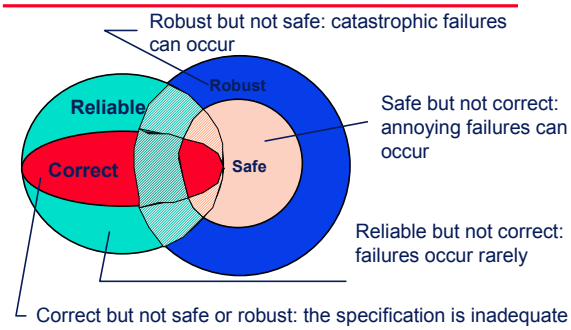
A framework for software testing

- Software qualities: identification of dependability properties
- Validation versus verification: identification of different types of activities
- Undecidability issues: why is software testing so difficult?
- Impact of type of software on testing
- The principles underlying software testing

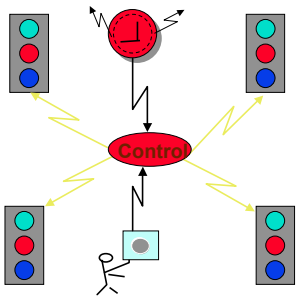
Software Qualities



Dependability Properties



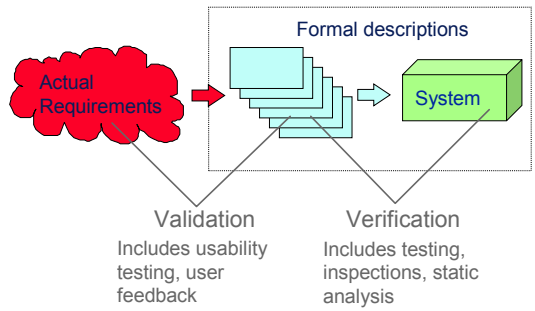
Example: Traffic light (USA version)



Correctness, reliability: *let traffic pass according to correct pattern and central scheduling*

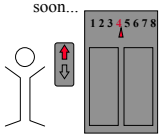
Robustness, safety: *Provide degraded function when possible; never signal conflicting greens.*

Validation vs. Verification



Verification or validation depend on the specification

- Unverifiable (but validatable) spec: ... if a user presses a request button at floor i, an available elevator must arrive at floor i soon...

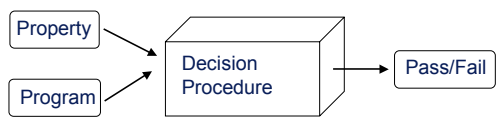


Example: elevator response

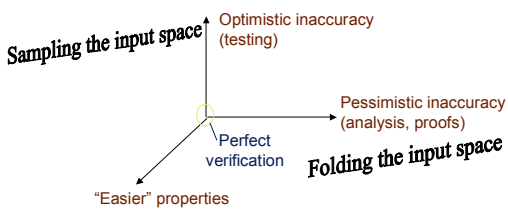
Verifiable spec: ... if a user presses a request button at floor i, an available elevator must arrive at floor i within 30 seconds...

You can't ~~always~~ *ever* get what you want

- Correctness properties are undecidable
  - the halting problem can be embedded in almost every property of interest



Getting what you need ...



We must make the problem of verification "easier" by permitting some kind of inaccuracy

Easier Properties - Example: Unmatched Semaphore Operations

```
if ( .... ) {
   ...
   lock(S);
}
...
if ( ... ) {
   ...
   unlock(S);
}
```

Static checking for match is necessarily inaccurate ...

... so Java prescribes a more restrictive, but statically checkable construct.

```
synchronized(S) {
        ...
        ...
}
```

Impact of the type of software on testing

- The type of software and its characteristics impact testing in different ways:
  - different emphasis may be given to the same properties
  - different (new) properties may be required
  - different (new) testing techniques may be needed

Different emphasis on the same properties

- Dependability requirements:
  - differ radically between
    - » Safety-critical applications
      - flight control systems have strict safety requirements
      - telecommunication systems have strict robustness requirements
    - » Mass-market products
      - dependability is less important than time to market
  - can vary within the same class of products:
    - » reliability and robustness are key issues for multi-user operating systems (e.g., UNIX)
    - » less important for single users operating systems (e.g., Windows or MacOS)

Different type of software may require different properties

- Timing properties
  - deadline satisfaction is a key issue for real time systems, but can be irrelevant for other systems
  - performance is important for many applications, but is the main issue for hard-real-time systems
- Synchronization properties
  - absence of deadlock is important for concurrent or distributed systems, not an issue for other systems
- External properties
  - user friendliness is an issue for GUI, irrelevant for embedded controllers

Different properties require different testing techniques

- Performance can be analyzed using statistical techniques, but deadline satisfaction requires exact computation of execution times
- Reliability can be checked with statistical testing techniques, correctness can be checked with weakest precondition computation (to prove the absence of faults)

Different testing techniques for checking the same properties for different software

- Test selection criteria based on structural coverage are different for
  - procedural software (statement, branch, path,…)
  - object oriented software (coverage of combination of polymorphic calls and dynamic bindings,…)
  - concurrent software (coverage of concurrent execution sequences,…)
  - mobile software

Principles

- Principles underlying effective software testing and analysis techniques include:
  - Sensitivity: better to fail every time than sometimes
  - Redundancy: make intentions explicit
  - Partitioning: divide and conquer
  - Restriction: make the problem easier
  - Feedback: tune the testing process

Sensitivity: Better to fail every time than sometimes

- Consistency helps:
  - a test selection criterion works better if every selected test provides the same result, i.e., if the program fails with one of the selected tests, it fails with all of them (reliable criteria)
  - Sometimes useful to turn off nondeterminism

Redundancy: make intentions explicit

- Redundant checks can increase the probability of catching specific faults early or more efficiently.
  - Static type checking is redundant with respect to dynamic type checking, but it can reveal many type mismatches earlier and more efficiently.
  - Validation of requirement specifications is redundant with respect to validation of the final software, but can reveal errors earlier and more efficiently.
  - Testing and proof of properties are redundant, but are often used together to increase confidence

Partitioning: divide and conquer

- Hard testing and verification problems can be handled by suitably partitioning the input space:
  - both structural and functional test selection criteria identify suitable partitions of code or specifications (partitions drive the sampling of the input space)

Restriction: make the problem easier

- Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems
  - A weaker spec may be easier to check: it is impossible (in general) to show that pointers are used correctly, but the simple Java requirement that pointers are initialized before use is simple to enforce.
  - A stronger spec may be easier to check: it is impossible (in general) to show that type errors do not occur at run-time in a dynamically typed language, but statically typed languages impose stronger restrictions that are easily checkable.

Feedback: tune the development process

- Learning from experience:
  - checklists are built on the basis of errors revealed in the past
  - error taxonomies can help in building better test selection criteria

Outline

- A framework for testing and analysis
- Overview of software testing
- Unit testing
- Summary

Testing

- Testing = executing the program on a sample of input data
- dynamic technique: programs must be executed
- optimistic inaccuracy: the program is executed on a (very small) subset on input data, the behavior of the program on every input is assumed to be consistent with the examined behaviors

Goals of testing

- Find faults ("Debug" Testing):
  - A Test is successful if the program fails
    (Goodeneogh, Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Jan. 85)
- Provide confidence (Acceptance Testing)
  - of reliability
  - of (probable) correctness
  - of detection (therefore absence) of particular faults

Testing is not a phase

- Quality assessment and improvement activities are spread through the whole development cycle:
  - from requirements elicitation:
    - » identify qualities,
    - » acceptance test planning
  - to maintenance:
    - » regression test execution
    - » revision of regression test suites

Testing sub-activities

- Test case execution is only a (relatively small) part of the process
- Must also consider
  - Test case generation
  - Test result evaluation
- Planning is essential
  - To achieve early and continuous visibility
  - To choose appropriate techniques at each stage
  - To build a testable product
  - To coordinate complementary analysis and testing

Granularity levels

- Acceptance testing
  - the software behavior is compared with end user requirements
- System testing
  - the software behavior is compared with the requirements specifications
- Integration testing
  - checking the behavior of module cooperation
- Unit testing
  - checking the behavior of single modules
- Regression testing
  - checking the behavior of new releases

Testing activities before coding

- Planning
  - acceptance test planning (requirements elicitation)
  - system test planning (requirements specifications)
  - Integration & unit test planning (architectural design)
- Generation
  - create functional system tests (requirement specifications)
  - generate test oracles (detailed design)
  - generate black box unit tests (detailed design)

Testing activities after coding

- Generation
  - create scaffolding (units)
- Execution
  - unit test execution (units)
  - integration test execution (subsystems)
  - system test execution (system)
  - acceptance test execution (system)
  - regression test execution (system)
- Measuring
  - coverage analysis (unit coding)
- Generation
  - delivery regression test suites (integration and delivery)
  - revise regression tests (maintenance)

The scaffolding problem

- How to provide the environment for executing the tests
- scaffolding is extremely important for unit and integration testing
- scaffolding may require substantial coding effort
- good scaffolding is an important step toward efficient regression testing

The oracle problem

- How to inspect the results of executing test and reveal failures
- Oracles are required at each stage of testing
- Automated test oracles required when running many tests
- Oracles are difficult to design - no universal recipe

The test case generation problem

- How to generate test data
  - Partition testing: divide program in (quasi-) equivalence classes
  - random
  - functional (black box)
    - » based on specifications
  - structural (white box)
    - » based on code
  - fault based
    - » based on classes of faults

White vs black box

- Black box
  - depends on the specification notation
  - scales up (different techniques at different granularity levels)
  - it cannot reveal code coverage problems (same specification implemented with different modules)
- White box
  - based on control or data flow coverage
  - does not scale up (mostly applicable at unit and integration testing level)
  - cannot reveal missing path errors (part of the specification that is not implemented)

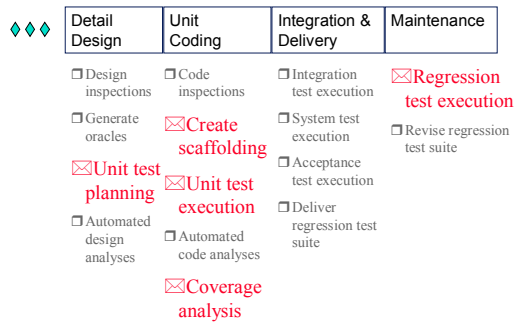
The termination problem

- How to decide when to stop testing
  - A hard managerial problem
  - when resources (time and budget) are over
    - » no information about the efficacy of testing
    - » BUT… resource constraints must be taken into account
  - When some coverage is reached
    - » no assurance of software quality
    - » it can be a reasonable and objective criterion
    - » It can be (partially) automated

Outline

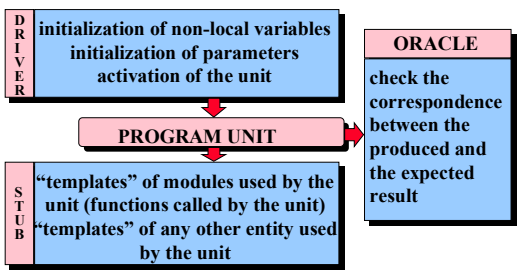
- A framework for testing and analysis
- Overview of software testing
- Unit testing
- Summary

Unit Testing: Main Activities

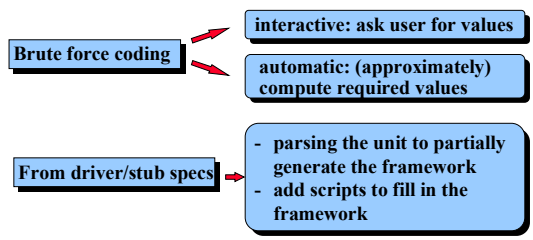


Create Scaffolding

- Goal: set up the environment for executing tests



Generate Drivers and Stubs



Test-case Generation from Natural Language

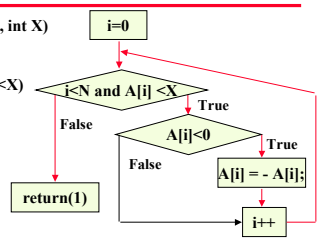
- cannot be automated
- some structure (e.g., organization standards) can help
- guidelines to increase confidence level and reduce discretionality: at least on test case for each:
  - subsets of "valid" homogeneous data
  - "non valid" (combinations of) data
  - boundary data
  - specific data (treated independently, error prone,...)

Structural Coverage Testing

- Adequacy criteria
  - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) coverage
  - Condition coverage
- Attempted compromise between the impossible and the inadequate

Statement Coverage

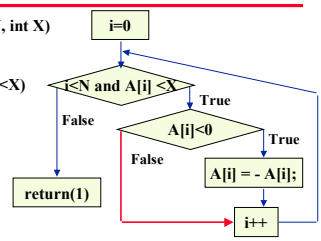
```
int select(int A[], int N, int X)
{
  int i=0;
  while (i<N and A[i] <X)
  {
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



One test datum (N=1, A[0]=-7, X=9) is enough to guarantee statement coverage of function select
Faults in handling positive values of A[i] would not be revealed

Branch Coverage

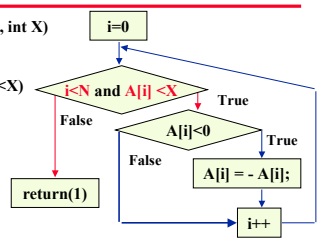
```
int select(int A[], int N, int X)
{
  int i=0;
  while (i<N and A[i] <X)
  {
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



We must add a test datum (N=1, A[0]=7, X=9) to cover branch False of the if statement. Faults in handling positive values of A[i] would be revealed. Faults in exiting the loop with condition A[i] <X would not be revealed

Condition Coverage

```
int select(int A[], int N, int X)
{
  int i=0;
  while (i<N and A[i] <X)
  {
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



Both conditions (i<N), (A[i]<X) must be false and true for different tests. In this case, we must add tests that cause the while loop to exit for a value greater than X. Faults that arise after several iterations of the loop would not be revealed.

The Budget coverage criterion

- Industry's 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: Adequacy criteria answer the wrong question. Selection is more important.
  - Implication 2: Practical comparison of approaches must consider the cost of test case selection

Regression Testing

- Testing a new version (release): how can we minimize effort using results of testing of previous versions?
- On a previous release:
  - save scaffoldings (drivers,stubs,oracles)
  - record test cases (<inputs,outputs>)
- On the new release:
  - keep track of changes
  - evaluate impact of changes

Outline

- A framework for testing and analysis
- Overview of software testing
- Unit testing
- Summary

Testing is Not a Phase

- Quality assessment and improvement activities must be spread through the whole development cycle
- Planning is essential
  - To achieve early and continuous visibility
  - To choose appropriate techniques at each stage
  - To build a testable product
  - To coordinate complementary analysis and testing techniques

Complementary Techniques

- There are no silver bullets
  - Different techniques depending on available artifacts
  - Different techniques depending on properties to be ascertained
  - Different techniques depending on acceptable cost and degree of assurance

Automation

- Automation is sometimes necessary
  - For some automated static analyses
  - To accurately check large sets of test results
- ... and often useful
  - For cost-effective regression testing
  - To monitor coverage, generate some tests automatically
- But some techniques are practical even without tools
  - Inspections of all artifacts, compilation of checklists, functional test case creation