Testing and Debugging

(Lecture 11)

Dr. R. Mall

Organization of this lecture

- Important concepts in program testing
- Black-box testing:
  - equivalence partitioning
  - boundary value analysis
- White-box testing
- Debugging
- Unit, Integration, and System testing
- Summary

Testing

- The aim of testing is to identify all defects in a software product.
- However, in practice even after thorough testing:
  - one cannot guarantee that the software is error-free.

Testing

- The input data domain of most software products is very large:
  - it is not practical to test the software exhaustively with each input data value.

Testing

- Testing does however expose many errors:
  - testing provides a practical way of reducing defects in a system
  - increases the users' confidence in a developed system.

Testing

- Testing is an important development phase:
  - requires the maximum effort among all development phases.
- In a typical development organization:
  - maximum number of software engineers can be found to be engaged in testing activities.

Testing

- Many engineers have the wrong impression:
  - testing is a secondary activity
  - it is intellectually not as stimulating as the other development activities, etc.

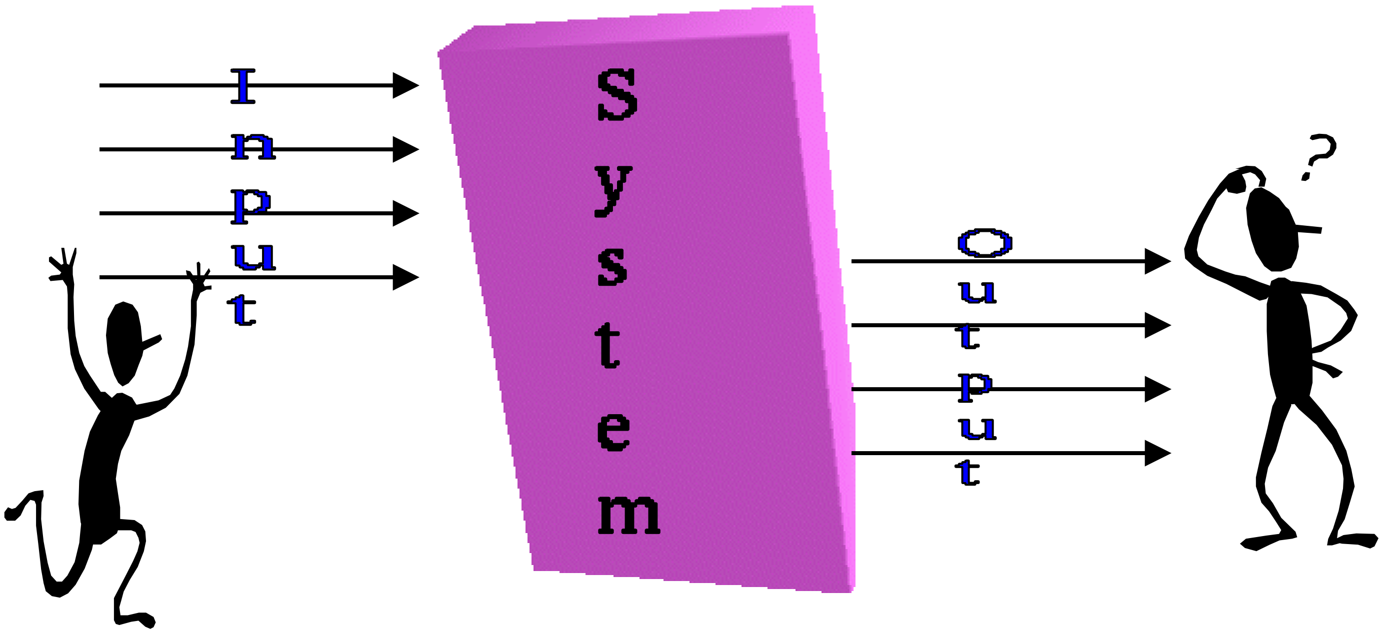
Testing

- Testing a software product is in fact:
  - as much challenging as initial development activities such as specification, design, and coding.
- Also, testing involves a lot of creative thinking.

How do you test a program?

- Input test data to the program.
- Observe the output:
  - Check if the program behaved as expected.

How do you test a system?



How do you test a system?

- If the program does not behave as expected:
  - note the conditions under which it failed.
  - later debug and correct.

Error, Faults, and Failures

- A failure is a manifestation of an error (aka defect or bug or fault).

Faults & Failure

- **Failure:** A software failure occurs if the behavior of the s/w is different from expected/specified.
- **Fault:** cause of software failure
- Fault = bug = defect
- Failure implies presence of defects
- A defect has the potential to cause failure.
- Definition of a defect is environment, project specific

Role of Testing

- Identify defects remaining after the review processes!
- Reviews are human processes - can not catch all defects
- There will be requirement defects, design defects and coding defects in code
- **Testing:**
  - Detects defects
  - plays a critical role in ensuring quality.

Error, Faults, and Failures

- A fault is an incorrect state entered during program execution:
  - a variable value is different from what it should be.
  - A fault may or may not not lead to a failure.

Test cases and Test suites

- Test a software using a set of carefully designed test cases:
  - the set of all test cases is called the test suite

Test cases and Test suites

- A test case is a triplet [I,S,O]
  - I is the data to be input to the system,
  - S is the state of the system at which the data will be input,
  - O is the expected output of the system.

Verification versus Validation

- Verification is the process of determining:
  - whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining
  - whether a fully developed system conforms to its SRS document.

Verification versus Validation

- Verification is concerned with phase containment of errors,
  - whereas the aim of validation is that the final product be error free.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - input data domain is extremely large.
- Design an optimal test suite:
  - of reasonable size and
  - uncovers as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
  - many test cases would not contribute to the significance of the test suite,
  - would not detect errors not already being detected by other test cases in the suite.
- Number of test cases in a randomly selected test suite:
  - not an indication of effectiveness of testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - does not mean that many errors in the system will be uncovered.
- Consider an example for finding the maximum of two integers x and y.

Design of Test Cases

- The code has a simple programming error:
- If (x>y) max = x;
  else max = x;
- test suite {(x=3,y=2);(x=2,y=3)} can detect the error,
- a larger test suite {(x=3,y=2);(x=4,y=3);(x=5,y=1)} does not detect the error.

Design of Test Cases

- Systematic approaches are required to design an optimal test suite:
  - each test case in the suite should detect different errors.

Design of Test Cases

- There are essentially two main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach

Black-box Testing

- Test cases are designed using only functional specification of the software:
  - without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as functional testing.

White-box Testing

- Designing white-box test cases:
  - requires knowledge about the internal structure of software.
  - white-box testing is also called structural testing.

Black-box Testing

- There are essentially two main approaches to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

Equivalence Class Partitioning

- Input values to a program are partitioned into equivalence classes.
- Partitioning is done such that:
  - program behaves in similar ways to every input value belonging to an equivalence class.

Why define equivalence classes?

- Test the code with just one representative value from each equivalence class:
  - as good as testing using any other values from the equivalence classes.

Equivalence Class Partitioning

- How do you determine the equivalence classes?
  - examine the input data.
  - few general guidelines for determining the equivalence classes can be given

Equivalence Class Partitioning

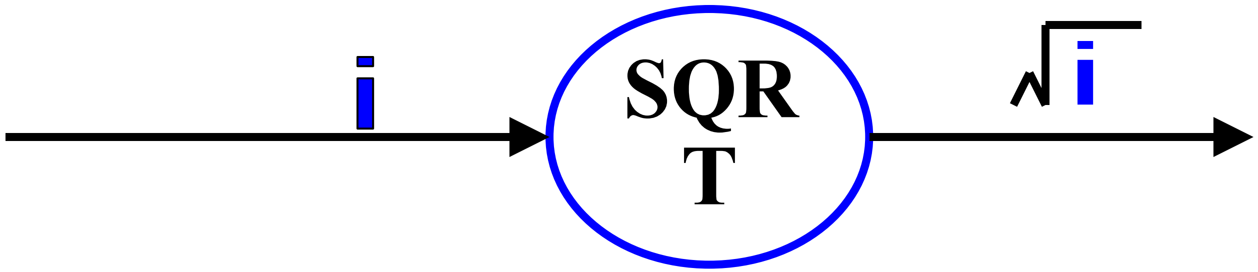
- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - one valid and two invalid equivalence classes are defined.

Equivalence Class Partitioning

- If input is an enumerated set of values:
  - e.g. {a,b,c}
  - one equivalence class for valid input values
  - another equivalence class for invalid input values should be defined.

Example

- A program reads an input value in the range of 100 and 1500:
  - computes the square root of the input number

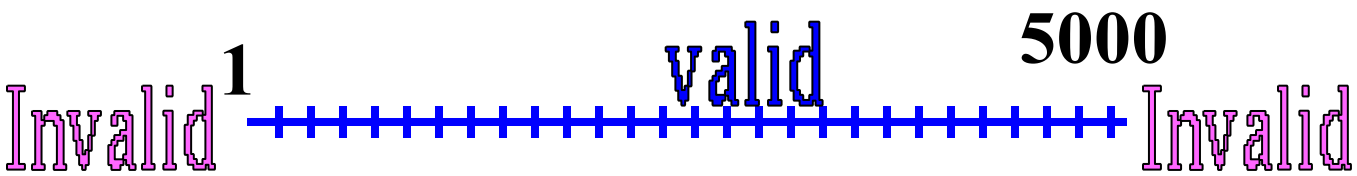


Example (cont.)

- There are three equivalence classes:
  - the set of negative integers,
  - set of integers in the range of 1 and 5000,
  - integers larger than 5000.

Example (cont.)

- The test suite must include:
  - representatives from each of the three equivalence classes:
  - a possible test suite can be: {-5,500,6000}.



Boundary Value Analysis

- Some typical programming errors occur:
  - at boundaries of equivalence classes
  - might be purely due to psychological factors.
- Programmers often fail to see:
  - special processing required at the boundaries of equivalence classes.

Boundary Value Analysis

- Programmers may improperly use < instead of <=
- Boundary value analysis:
  - select test cases at the boundaries of different equivalence classes.

Example

- For a function that computes the square root of an integer in the range of 1 and 5000:
  - test cases must include the values: {0,1,5000,5001}.

Testing

- Software products are tested at three levels:
  - Unit testing
  - Integration testing
  - System testing

Unit testing

- During unit testing, modules are tested in isolation:
  - If all modules were to be tested together:
    - it may not be easy to determine which module has the error.

Unit testing

- Unit testing reduces debugging effort several folds.
  - Programmers carry out unit testing immediately after they complete the coding of a module.

Integration testing

- After different modules of a system have been coded and unit tested:
  - modules are integrated in steps according to an integration plan
  - partially integrated system is tested at each integration step.

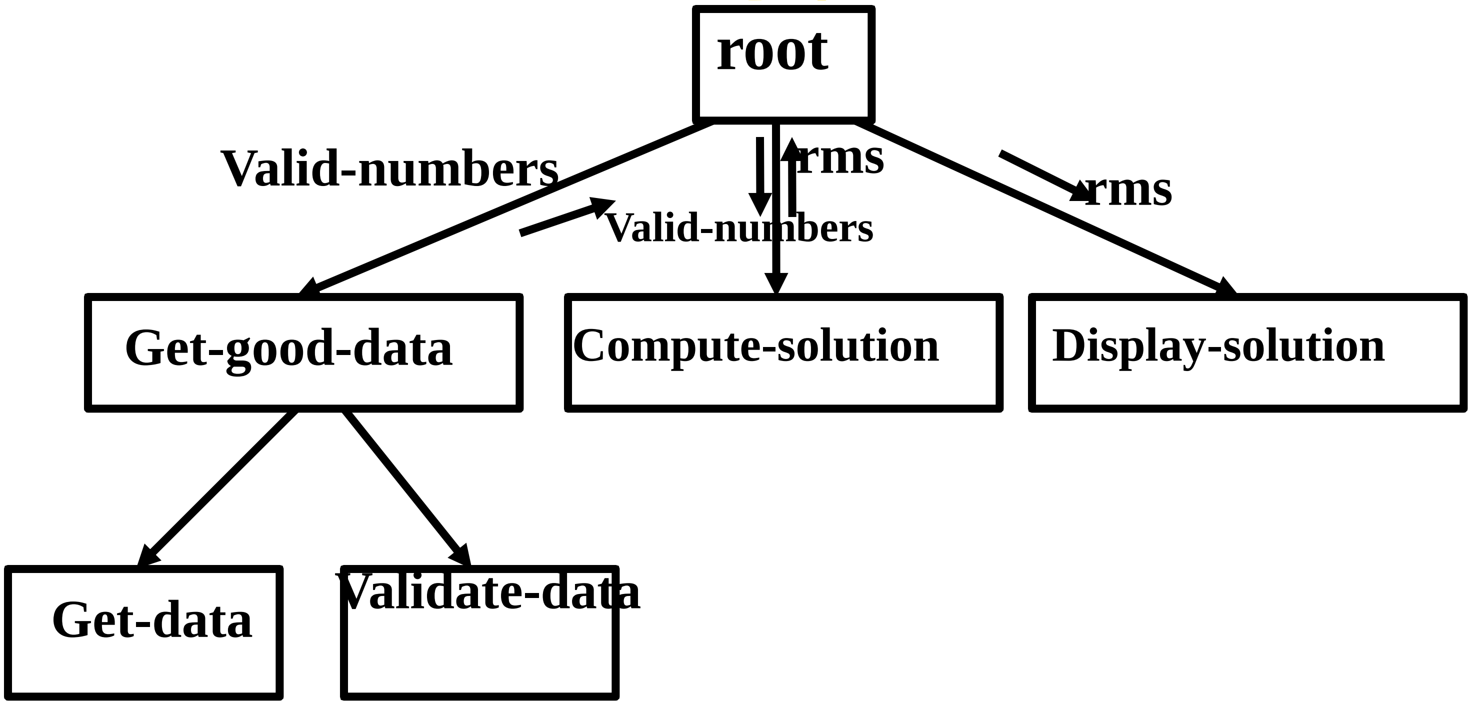
System Testing

- System testing involves:
  - validating a fully developed system against its requirements.

Integration Testing

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach

Example Structured Design



Big bang Integration Testing

- Big bang approach is the simplest integration testing approach:
  - all the modules are simply put together and tested.
  - this technique is used only for very small systems.

Big bang Integration Testing

- Main problems with this approach:
  - if an error is found:
    - it is very difficult to localize the error
    - the error may potentially belong to any of the modules being integrated.
  - debugging errors found during big bang integration testing are very expensive to fix.

Bottom-up Integration Testing

- Integrate and test the bottom level modules first.
- A disadvantage of bottom-up testing:
  - when the system is made up of a large number of small subsystems.
  - This extreme case corresponds to the big bang approach.

Top-down integration testing

- Top-down integration testing starts with the main routine:
  - and one or two subordinate routines in the system.
- After the top-level 'skeleton’ has been tested:
  - immediate subordinate modules of the 'skeleton’ are combined with it and tested.

Mixed integration testing

- Mixed (or sandwiched) integration testing:
  - uses both top-down and bottom-up testing approaches.
  - Most common approach

Integration Testing

- In top-down approach:
  - testing waits till all top-level modules are coded and unit tested.
- In bottom-up approach:
  - testing can start only after bottom level modules are ready.

System Testing

- There are three main kinds of system testing:
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing

Alpha Testing

- System testing is carried out by the test team within the developing organization.

Beta Testing

- System testing performed by a select group of friendly customers.

Acceptance Testing

- System testing performed by the customer himself:
  - to determine whether the system should be accepted or rejected.

Stress Testing

- Stress testing (aka endurance testing):
  - impose abnormal input to stress the capabilities of the software.
  - Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

How many errors are still remaining?

- Seed the code with some known errors:
  - artificial errors are introduced into the program.
  - Check how many of the seeded errors are detected during testing.

Error Seeding

- Let:
  - N be the total number of errors in the system
  - n of these errors be found by testing.
  - S be the total number of seeded errors,
  - s of the seeded errors be found during testing.

Error Seeding

- $n/N = s/S$
- $N = S\, n/s$
- remaining defects:

  $N - n = n \;((S - s)/ s)$

Example

- 100 errors were introduced.
- 90 of these errors were found during testing
- 50 other errors were also found.
- Remaining errors= 50 (100-90)/90 = 6

Error Seeding

- The kind of seeded errors should match closely with existing errors:
  - However, it is difficult to predict the types of errors that exist.
- Categories of remaining errors:
  - can be estimated by analyzing historical data from similar projects.

Summary

- Exhaustive testing of almost any non-trivial system is impractical.
  - we need to design an optimal test suite that would expose as many errors as possible.

Summary

- If we select test cases randomly:
  - many of the test cases may not add to the significance of the test suite.
- There are two approaches to testing:
  - black-box testing
  - white-box testing.

Summary

- Black box testing is also known as functional testing.
- Designing black box test cases:
  - requires understanding only SRS document
  - does not require any knowledge about design and code.
- Designing white box testing requires knowledge about design and code.

Summary

- We discussed black-box test case design strategies:
  - equivalence partitioning
  - boundary value analysis
- We discussed some important issues in integration and system testing.