Software Testing

Definition

- Glen Myers
  - Testing is the process of executing a program with the intent of finding *errors*
- Paul Jorgensen
  - Testing is obviously concerned with *errors, faults, failures and incidents*. A test is the act of exercising software with *test cases* with an objective of
    - Finding failure
    - Demonstrate correct execution

Terminology

- Error
  - Represents mistakes made by people
- Fault
  - Is result of error. May be categorized as
    - Fault of Commission – we enter something into representation that is incorrect
    - Fault of Omission – Designer can make error of omission, the resulting fault is that something is missing that should have been present in the representation

Cont…

- Failure
  - Occurs when fault executes.
- Incident
  - Behavior of fault. An incident is the symptom(s) associated with a failure that alerts user to the occurrence of a failure
- Test case
  - Associated with program behavior. It carries set of input and list of expected output

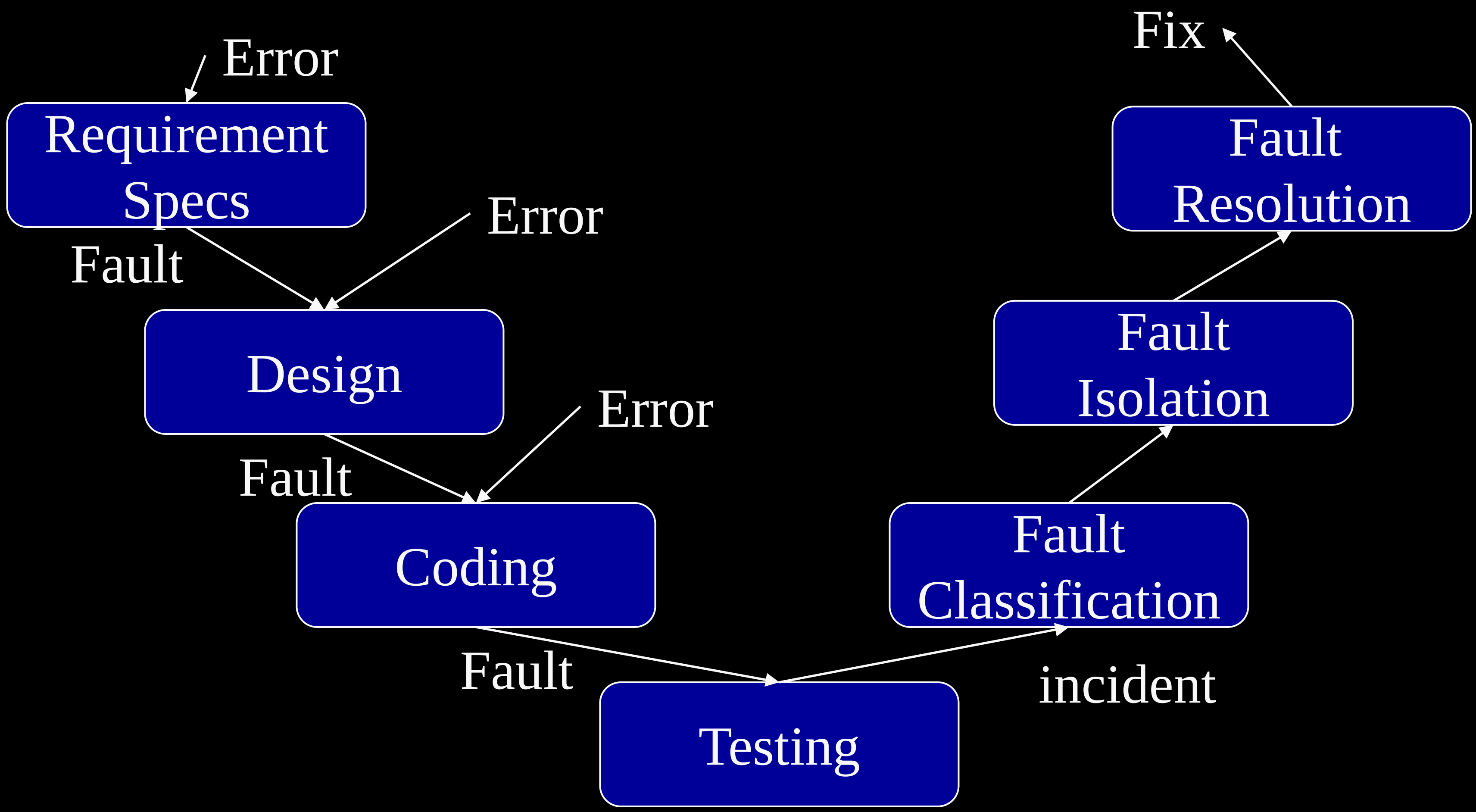
Cont…

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

Verification versus Validation

- Verification is concerned with phase containment of errors
- Validation is concerned about the final product to be error free

Testing - Life Cycle



Classification of Test

- There are two levels of classification
  - One distinguishes at granularity level
    - Unit level
    - System level
    - Integration level
  - Other classification (mostly for unit level) is based on methodologies
    - Black box (Functional) Testing
    - White box (Structural) Testing

Unit testing

- Applicable to modular design
  - Unit testing inspects individual modules
- Locate error in smaller region
  - In an integrated system, it may not be easier to determine which module has caused fault
  - Reduces debugging efforts

Test cases and Test suites

- Test case is a triplet [I, S, O] where
  - I is input data
  - S is state of system at which data will be input
  - O is the **expected** output
- Test suite is set of all test cases

Need for designing test cases

- Almost every non-trivial system has an extremely large input data domain thereby making exhaustive testing impractical
- If randomly selected then test case may loose significance since it may expose an already detected error by some other test case

Design of test cases

- Number of test cases do not determine the effectiveness
- Each test case should detect different errors

Black box testing

- Equivalence class partitioning
- Boundary value analysis
- Comparison testing
- Orthogonal array testing
- Decision Table based testing
  - Cause Effect Graph

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

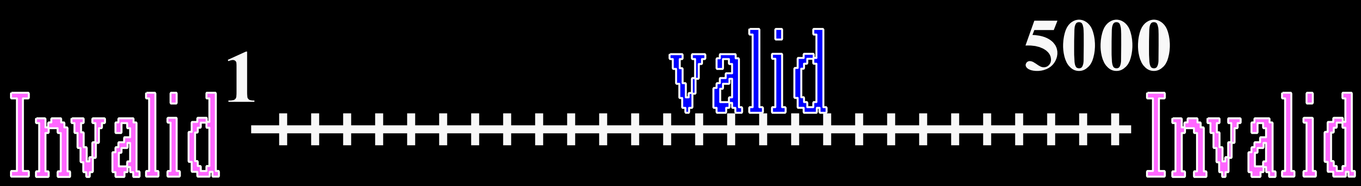
- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - one valid and two invalid equivalence classes are defined.

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 (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}.

White-Box Testing

- Statement coverage
- Branch coverage
- Path coverage
- Condition coverage
- Mutation testing
- Data flow-based testing

Statement Coverage

- Statement coverage methodology:
  - design test cases so that every statement in a program is executed at least once.
- The principal idea:
  - unless a statement is executed, we have no way of knowing if an error exists in that statement

Example

Euclid's GCD Algorithm

```
int f1(int x, int y){
while (x != y){
    if (x>y) then
        x=x-y;
    else y=y-x;
 }
return x;        }
```

Euclid's GCD computation algorithm

- By choosing the test set {(x=3,y=3),(x=4,y=3), (x=3,y=4)}
  - all statements are executed at least once.

Branch Coverage

- Test cases are designed such that:
  - different branch conditions is given true and false values in turn.
- Branch testing guarantees statement coverage:
  - a stronger testing compared to the statement coverage-based testing.

Example

- Test cases for branch coverage can be:

  {(x=3,y=3), (x=4,y=3), (x=3,y=4)}

Condition Coverage

- Test cases are designed such that:
  - each component of a composite conditional expression given both true and false values.
- Example
  - Consider the conditional expression ((c1.and.c2).or.c3):
  - Each of c1, c2, and c3 are exercised at least once i.e. given true and false values.

Condition coverage

- Consider a Boolean expression having n components:
  - for condition coverage we require 2^n test cases.
- practical only if n (the number of component conditions) is small.

Path Coverage

- Design test cases such that:
  - all linearly independent paths in the program are executed at least once.
- Defined in terms of
  - control flow graph (CFG) of a program.

Control flow graph (CFG)

- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

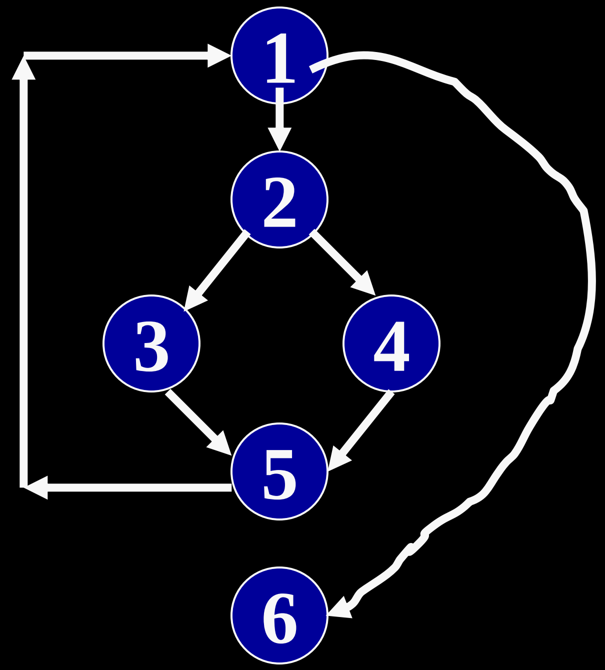
How to draw Control flow graph?

- Number all the statements of a program.
- Numbered statements:
  - represent nodes of the control flow graph.
- An edge from one node to another node exists:
  - if execution of the statement representing the first node can result in transfer of control to the other node.

Example

```
int f1(int x,int y){
1.   while (x != y){
2.       if (x>y) then
3.           x=x-y;
4.       else y=y-x;
5.    }
6.    return x;       }
```

Example Control Flow Graph



Path

- A path through a program:
  - A node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

Independent path

- Any path through the program:
  - introducing at least one new node that is not included in any other independent paths.
- It may be straight forward to identify linearly independent paths of simple programs. However For complicated programs it is not so easy to determine the number of independent paths.

McCabe's cyclomatic metric

- An upper bound:
  - for the number of linearly independent paths of a program
- Provides a practical way of determining:
  - the maximum number of linearly independent paths in a program.

McCabe's cyclomatic metric

- Given a control flow graph G, cyclomatic complexity V(G):
  - V(G)= E-N+2
    - N is the number of nodes in G
    - E is the number of edges in G

Example

- Cyclomatic complexity =
  $7 - 6 + 2 = 3$.

Cyclomatic complexity

- Another way of computing cyclomatic complexity:
  - determine number of bounded areas in the graph
    - Any region enclosed by a nodes and edge sequence.
- V(G) = Total number of bounded areas + 1

Example

- From a visual examination of the CFG:
  - the number of bounded areas is 2.
  - cyclomatic complexity = 2+1=3.

Cyclomatic complexity

- McCabe's metric provides:
  - a quantitative measure of estimating testing difficulty
  - Amenable to automation
- Intuitively,
  - number of bounded areas increases with the number of decision nodes and loops.

Cyclomatic complexity

- The cyclomatic complexity of a program provides:
  - a lower bound on the number of test cases to be designed
  - to guarantee coverage of all linearly independent paths.

Path testing

- The tester proposes initial set of test data using his experience and judgement.

Path testing

- A testing tool such as dynamic program analyzer, then may be used:
  - to indicate which parts of the program have been tested
  - the output of the dynamic analysis used to guide the tester in selecting additional test cases.

Data Flow-Based Testing

- Selects test paths of a program:
  - according to the locations of definitions and uses of different variables in a program.

Data Flow-Based Testing

- For a statement numbered S,
  - DEF(S) = {X/statement S contains a definition of X}
  - USES(S)= {X/statement S contains a use of X}
  - Example: 1: a=b; DEF(1)={a}, USES(1)={b}.
  - Example: 2: a=a+b; DEF(1)={a}, USES(1)={a,b}.

Mutation Testing

- The software is first tested:
  - using an initial testing method based on white-box strategies.
- After the initial testing is complete,
  - mutation testing is taken up.
- The idea behind mutation testing:
  - make a few arbitrary small changes to a program at a time.

Mutation Testing

- Each time the program is changed,
  - it is called a mutated program
  - the change is called a mutant.

Mutation Testing

- A mutated program:
  - tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - a mutant gives an incorrect result, then the mutant is said to be dead.

Mutation Testing

- If a mutant remains alive:
  - even after all test cases have been exhausted, the test suite is enhanced to kill the mutant.
- The process of generation and killing of mutants:
  - can be automated by predefining a set of primitive changes that can be applied to the program.

Mutation Testing

- The primitive changes can be:
  - altering an arithmetic operator,
  - changing the value of a constant,
  - changing a data type, etc.

Mutation Testing

- A major disadvantage of mutation testing:
  - computationally very expensive,
  - a large number of possible mutants can be generated.

Debugging

- Once errors are identified:
  - it is necessary identify the precise location of the errors and to fix them.
- Each debugging approach has its own advantages and disadvantages:
  - each is useful in appropriate circumstances.

Brute-force method

- This is the most common method of debugging:
  - least efficient method.
  - program is loaded with print statements
  - print the intermediate values
  - hope that some of printed values will help identify the error.

Symbolic Debugger

- Brute force approach becomes more systematic:
  - with the use of a symbolic debugger

Symbolic Debugger

- Using a symbolic debugger:
  - values of different variables can be easily checked and modified
  - single stepping to execute one instruction at a time
  - break points and watch points can be set to test the values of variables.

Backtracking

- This is a fairly common approach.
- Beginning at the statement where an error symptom has been observed:
  - source code is traced backwards until the error is discovered.

Backtracking

- Unfortunately, as the number of source lines to be traced back increases,
  - the number of potential backward paths increases
  - becomes unmanageably large for complex programs.

Program Slicing

- This technique is similar to back tracking.
- However, the search space is reduced by defining slices.
- A slice is defined for a particular variable at a particular statement:
  - set of source lines preceding this statement which can influence the value of the variable.

Debugging Guidelines

- Debugging usually requires a thorough understanding of the program design.
- Debugging may sometimes require full redesign of the system.
- A common mistake novice programmers often make:
  - not fixing the error but the error symptoms.

Debugging Guidelines

- Be aware of the possibility:
  - an error correction may introduce new errors.
- After every round of error-fixing:
  - regression testing must be carried out.

Program Analysis Tools

- An automated tool:
  - takes program source code as input
  - produces reports regarding several important characteristics of the program,
  - such as size, complexity, adequacy of commenting, adherence to programming standards, etc.

Program Analysis Tools

- Some program analysis tools:
  - produce reports regarding the adequacy of the test cases.
- There are essentially two categories of program analysis tools:
  - Static analysis tools
  - Dynamic analysis tools

Static Analysis Tools

- Static analysis tools:
  - assess properties of a program without executing it.
  - Analyze the source code
    - provide analytical conclusions.

Static Analysis Tools

- Whether coding standards have been adhered to?
  - Commenting is adequate?
- Programming errors such as:
  - Un-initialized variables
  - mismatch between actual and formal parameters.
  - Variables declared but never used, etc.

Static Analysis Tools

- Code walk through and inspection can also be considered as static analysis methods:
  - however, the term static program analysis is generally used for automated analysis tools.

Dynamic Analysis Tools

- Dynamic program analysis tools require the program to be executed:
  - its behaviour recorded.
  - Produce reports such as adequacy of test cases.

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

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.

Phased versus Incremental Integration Testing

- Integration can be incremental or phased.
- In incremental integration testing,
  - only one new module is added to the partial system each time.

Phased versus Incremental Integration Testing

- In phased integration,
  - a group of related modules are added to the partially integrated system each time.

Phased versus Incremental Integration Testing

- Phased integration requires less number of integration steps:
  - compared to the incremental integration approach.
- However, when failures are detected,
  - it is easier to debug if using incremental testing
    - since errors are very likely to be in the newly integrated module.

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 (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.

Performance Testing

- Addresses non-functional requirements.
  - May sometimes involve testing hardware and software together.
  - There are several categories of performance testing.

Stress testing

- Evaluates system performance
  - when stressed for short periods of time.
- Stress testing
  - also known as endurance testing.

Stress testing

- Stress tests are black box tests:
  - designed to impose a range of abnormal and even illegal input conditions
  - so as to stress the capabilities of the software.

Stress Testing

- If the requirements is to handle a specified number of users, or devices:
  - stress testing evaluates system performance when all users or devices are busy simultaneously.

Volume Testing

- Addresses handling large amounts of data in the system:
  - whether data structures (e.g. queues, stacks, arrays, etc.) are large enough to handle all possible situations
  - Fields, records, and files are stressed to check if their size can accommodate all possible data volumes.

Configuration Testing

- Analyze system behaviour:
  - in various hardware and software configurations specified in the requirements
  - sometimes systems are built in various configurations for different users

Compatibility Testing

- These tests are needed when the system interfaces with other systems:
  - check whether the interface functions as required.

Recovery Testing

- These tests check response to:
  - the loss of data, power, devices, or services
  - subject system to loss of resources
    - check if the system recovers properly.

Maintenance Testing

- Verify that:
  - all required artefacts for maintenance exist
  - they function properly

Documentation tests

- Check that required documents exist and are consistent:
  - user guides,
  - maintenance guides,
  - technical documents

Usability tests

- All aspects of user interfaces are tested:
  - Display screens
  - messages
  - report formats
  - navigation and selection problems

Environmental test

- These tests check the system’s ability to perform at the installation site.
- Requirements might include tolerance for
  - heat
  - humidity
  - chemical presence
  - portability
  - electrical or magnetic fields
  - disruption of power, etc.

Regression Testing

- Does not belong to either unit test, integration test, or system test.
  - In stead, it is a separate dimension to these three forms of testing.

Regression testing

- Regression testing is the running of test suite:
  - after each change to the system or after each bug fix
  - ensures that no new bug has been introduced due to the change or the bug fix.

Regression testing

- Regression tests assure:
  - the new system’s performance is at least as good as the old system
  - always used during phased system development.

IEEE Standard

- Test plan identifier
- Introduction
- Test Items
- Features to be tested
- Features not to be tested
- Approach
- Item pass/fail criteria
- Suspension criteria and resumption requirements

Cont…

- Test deliverables
- Testing tasks
- Environment needs
- Responsibilities
- Staffing and training needs
- Risk and contingencies
- Approvals

References

- Software Testing, A craftsman's approach
  - Paul Jorgensen
- Fundamental of Software Engineering
  - Rajib Mall
- Software Engineering, A practitioner's approach
  - Roger Pressman