Software Engineering

LECTURE-42

Software Testing and Validation Overview

Topics Covered

- Software Testing
- Test Plans
- Test plan Considerations
- IEEE 829
- Reality Check

What is Software Testing?

Several definitions:

"Testing is the process of establishing confidence that a program or system does what it is supposed to." by Hetzel 1973

"Testing is the process of executing a program or system with the intent of finding errors." by Myers 1979

"Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results." by Hetzel 1983

What is Software Testing?

- One of very important software development phases

- A software process based on well-defined software quality control and testing standards, testing methods, strategy, test criteria, and tools.

- Engineers perform all types of software testing activities to perform a software test process.

- The last quality checking point for software on its production line

Who does Software Testing?

- **Test manager**
  - **manage and control a software test project**
  - **supervise test engineers**
  - **define and specify a test plan**

- **Software Test Engineers and Testers**
  - **define test cases, write test specifications, run tests**

- **Independent Test Group**

- **Development Engineers**
  - **Only perform unit tests and integration tests**

- **Quality Assurance Group and Engineers**
  - **Perform system testing**
  - **Define software testing standards and quality control process**

The Test Plan

- **The Test Plan**
  - who
  - what
  - when
  - where
  - how

Test Plan Considerations

- What are the critical or most complex modules?
  - make sure they get integration tested first
  - probably deserve white-box attention
- Where have you had problems in the past?
- Third-Party delivered components?
- What training is required?
  - conducting formal reviews
  - use of testing tools
  - defect report logging

IEEE 829 - Standard for Software Test Documentation

Recommends 8 types of testing documents:

1. Test Plan
   - *next slide*
2. Test Design Specification
   - expected results, pass criteria, ...
3. Test Case Specification
   - test data for use in running the tests
4. Test Procedure Specification
   - how to run each test
5. Test Item Transmittal Report
   - reporting on when components have progressed from one stage of testing to the next
6. Test Log
7. Test Incident Report
   - for any test that failed, the actual versus expected result
8. Test Summary Report
   - management report

http://en.wikipedia.org/wiki/IEEE_829

Test Plan Contents (IEEE 829 format)

1. Test Plan Identifier
2. References
3. Introduction
4. Test Items
   see next slide
5. Software Risk Issues
6. Features to be Tested
7. Features not to be Tested
8. Approach
9. Item Pass/Fail Criteria
10. Suspension Criteria and Resumption Requirements
11. Test Deliverables
12. Remaining Test Tasks
13. Environmental Needs
14. Staffing and Training Needs
15. Responsibilities
16. Schedule
17. Planning Risks and Contingencies
18. Approvals
19. Glossary

http://en.wikipedia.org/wiki/Test_Plan

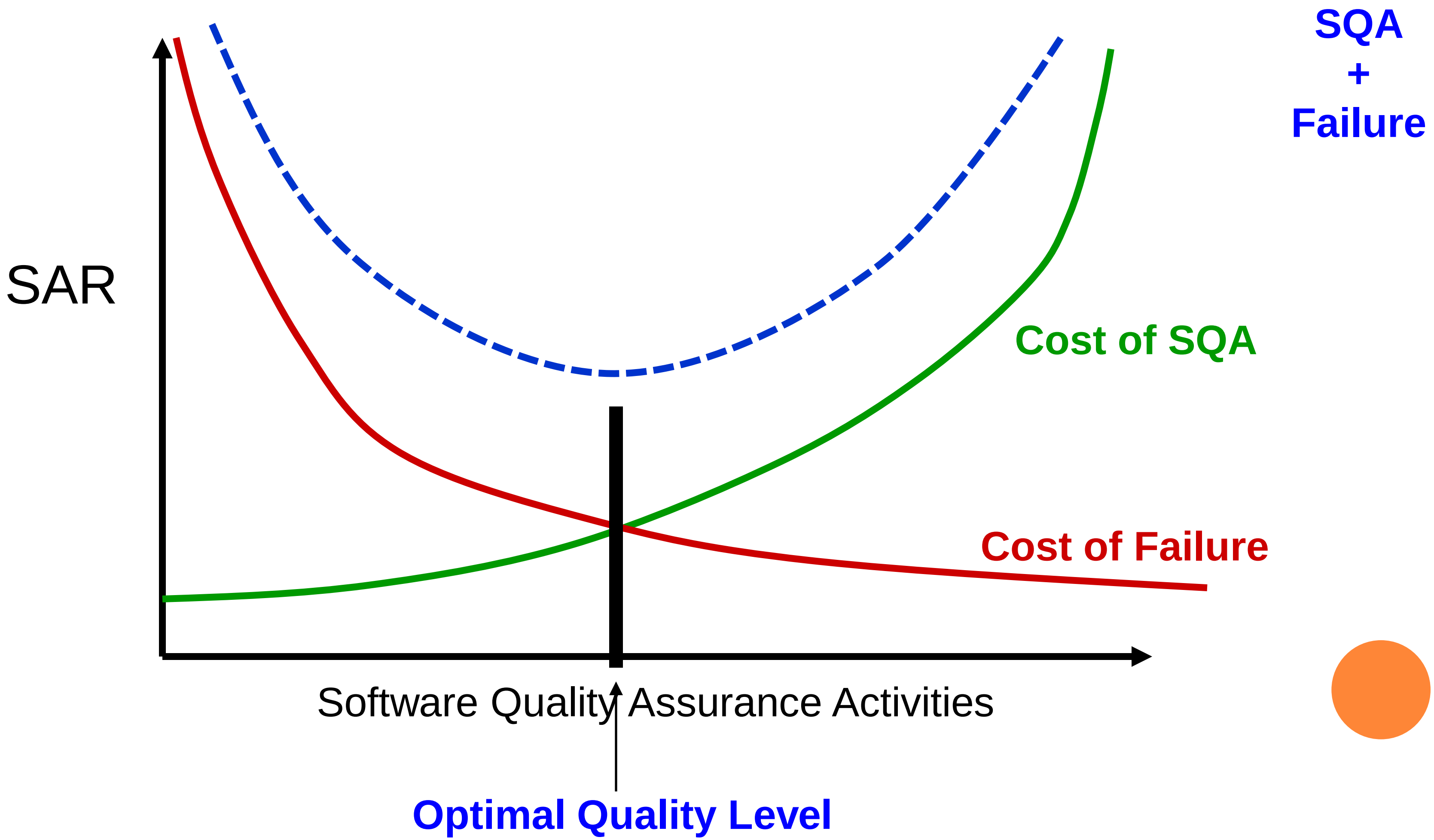
Test Items

- Requirements Specification
- Design
- Modules
- User/Operator Material
  - the user interface
  - User Guide
  - Operations Guide
- Features
  - response time, data accuracy, security, etc
- System Validation
  - alpha and beta testing

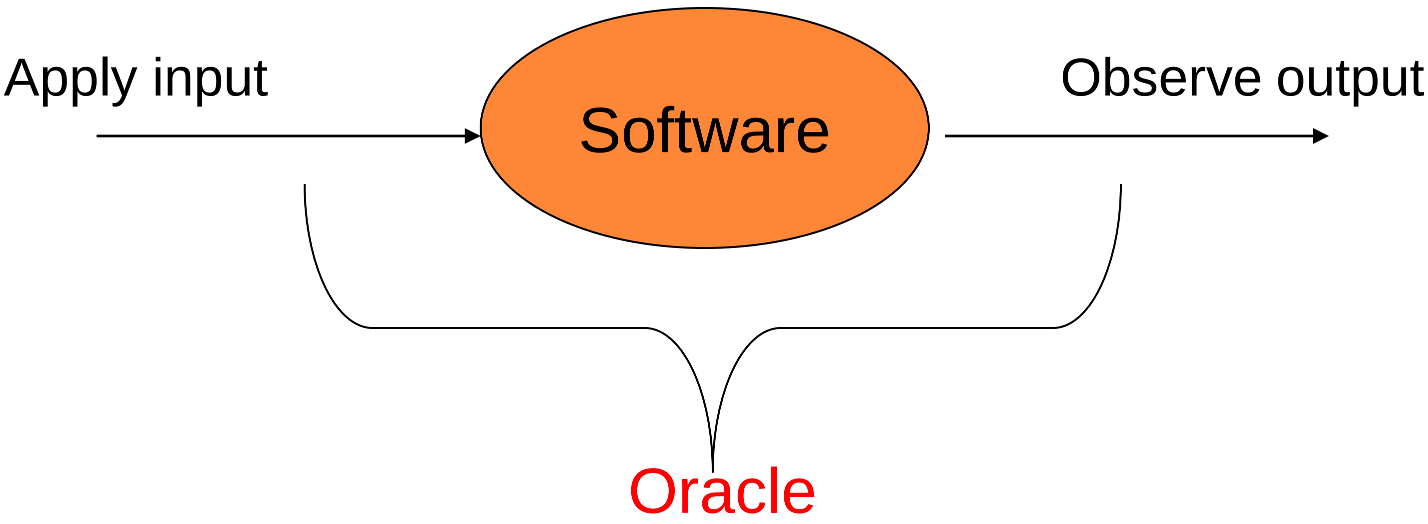
Based on IEEE 829

Reality Check

- ***When is more testing not cost effective?***



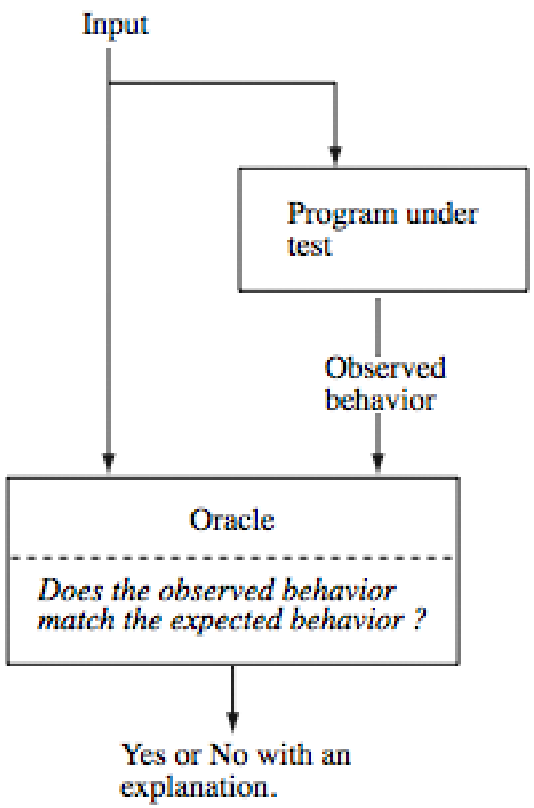
Testing



Validate the observed output against the expected output

Is the observed output the same as the expected output?

Oracle: Examples



- How to verify the output of a matrix multiplication?
- How to verify the output of a matrix inversion program?
- How to verify the output of a sorting algorithm?

Oracle: Example

A tester often assumes the role of an oracle and thus serves as human oracle.

How to verify the output of a matrix multiplication?

Hand calculation: the tester might input two matrices and check if the output of the program matches the results of hand calculation.

Oracles can also be programs. For example, one might use a matrix multiplication to check if a matrix inversion program has produced the correct result: $A \times A^{-1} = I$

How to verify the output of a sorting algorithm?

ORACLE: CONSTRUCTION

Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, requires the determination of input-output relationship.

In general, the construction of automated oracles is a complex undertaking.

Limitations of Testing

- 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)

Software Testing Principles

- Principle #1: Complete testing is impossible.

- Principle #2: Software testing is not simple activity.
  - Reasons:
    - Quality testing requires testers to understand a system/product completely
    - Quality testing needs adequate test set, and efficient testing methods
    - A very tight schedule and lack of test tools.

- Principle #3: Testing is risk-based.

- Principle #4: Testing must be planned.

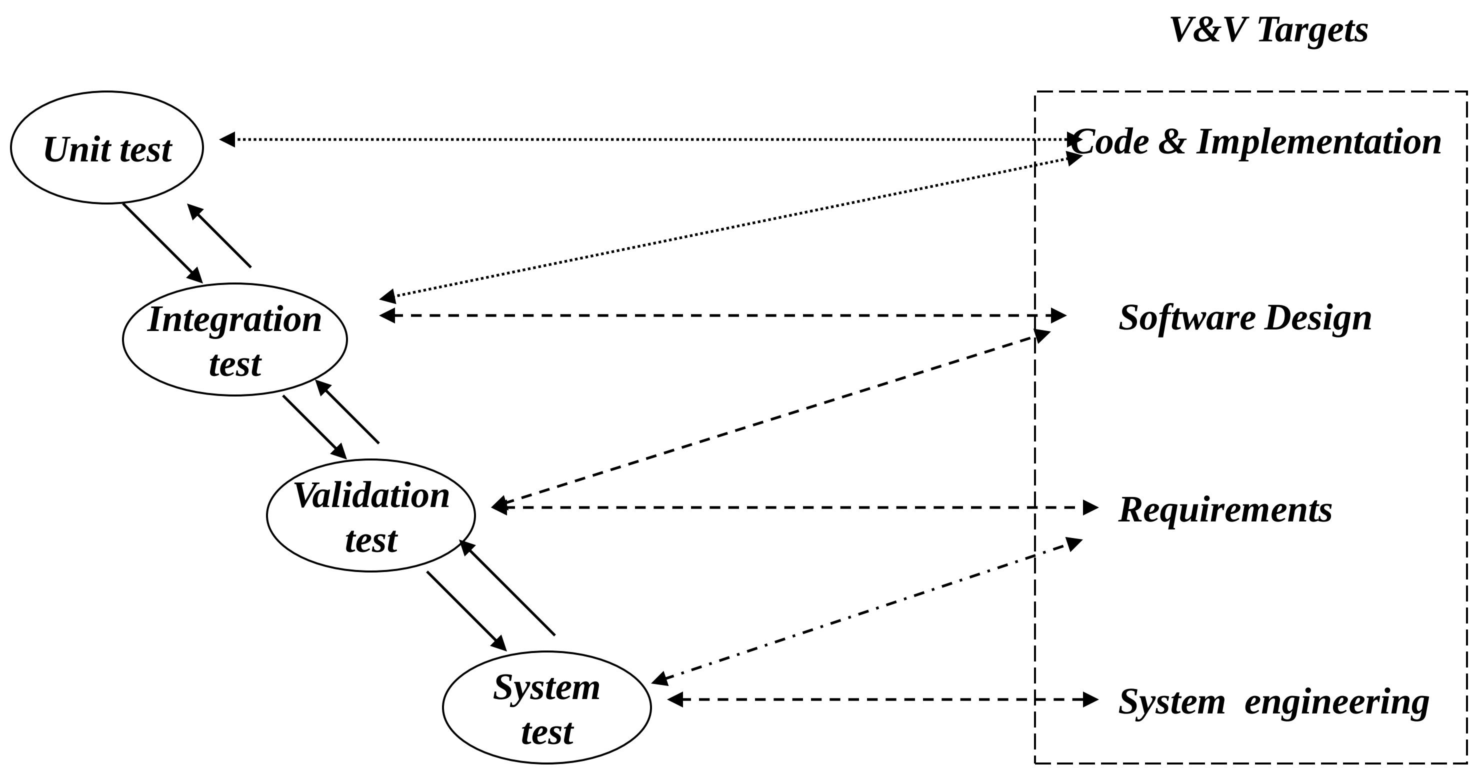
- Principle #5: Testing requires independence (SQA team).

- Principle #6: Quality software testing depends on:
    - Good understanding of software products and related domain application
    - Cost-effective testing methodology, coverage, test methods, and tools.
    - Good engineers with creativity, and solid software testing experience

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

Software Testing Process



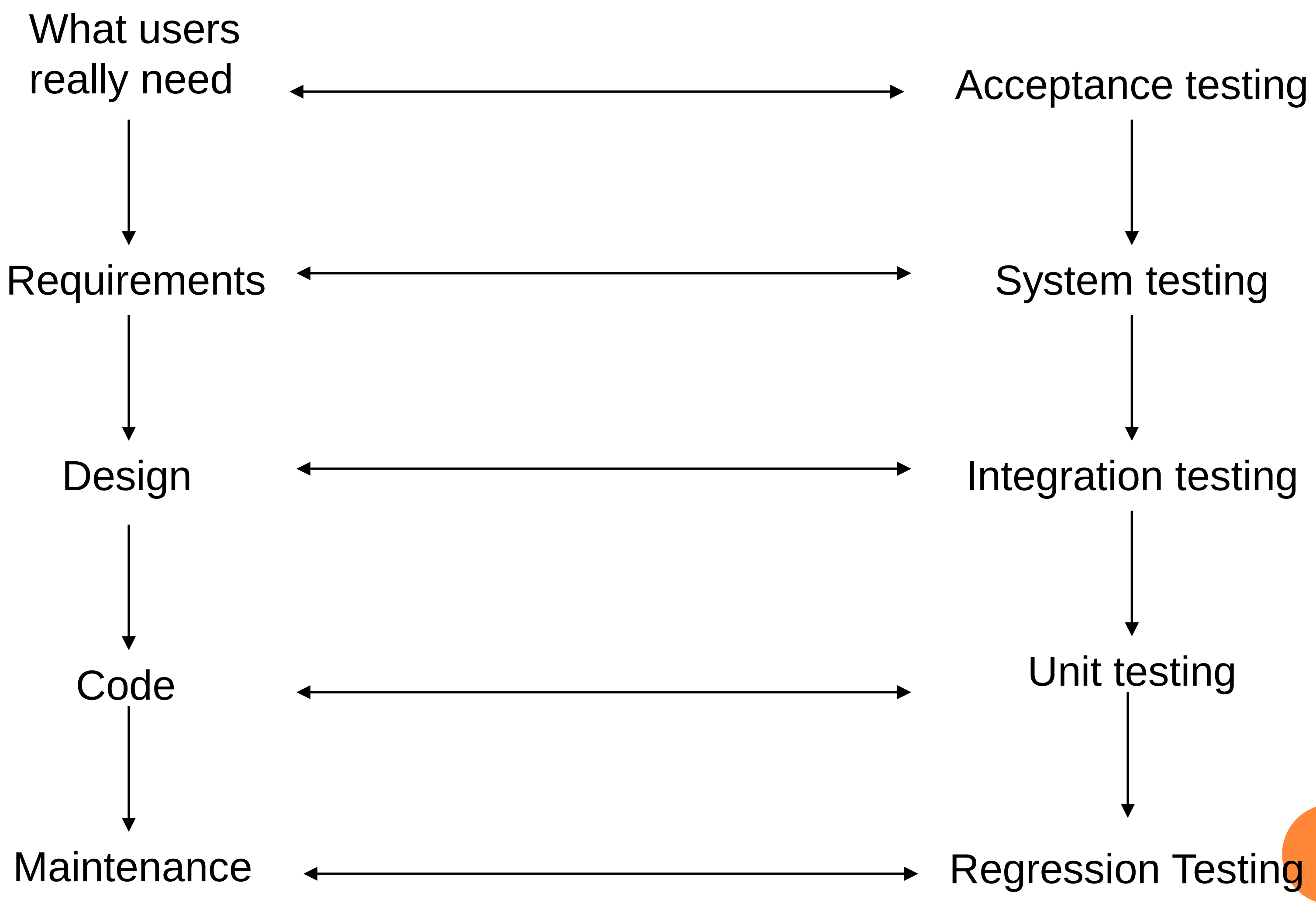
TESTING PROCESS GOALS

- Validation testing
  - To demonstrate to the developer and the system customer that the software meets its requirements;
  - A successful test shows that the system operates as intended.
- Verification – Defect testing
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification;
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Levels of Testing

- Component/Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing

Levels of Testing



Software Testing Process

Component/Unit Testing

Component Testing

- Testing of individual program components;
- Usually the responsibility of the component developer (except sometimes for critical systems);
- Tests are derived from the developer's experience.
- Require knowledge of code
  - High level of detail
  - Deliver thoroughly tested components to integration
- Stopping criteria
  - Code Coverage
  - Quality

Component Testing

- 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

Software Testing Process

INTEGRATION TESTING

Integration Testing

- Test assembled components
  - These must be tested and accepted previously
- Focus on interfaces
  - Might be interface problem although components work when tested in isolation
  - Might be possible to perform new tests

Integration Testing

- Strategies
  - Bottom-up, start from bottom and add one at a time
  - Top-down, start from top and add one at a time
  - Big-bang, everything at once
  - Functional, order based on execution

- Simulation of other components
  - Stubs receive output from test objects
  - Drivers generate input to test objects
  - Note that these are also SW, i.e., need testing etc.

Integration Testing

- There are two groups of software integration strategies:

  - Non Incremental software integration
  - Incremental software integration

  Non Incremental software integration:

  Big bang integration approach

  Incremental software integration:

  Top- down software integration
  Bottom-up software integration
  Sandwich integration

Integration Testing

- Involves building a system from its components and testing it for problems that arise from component interactions.

- Top-down integration
  - Develop the skeleton of the system and populate it with components. Use stubs to replace real components.
  - Two strategies: depth first and breadth first.

- Bottom-up integration
  - Integrate infrastructure components then add functional components. Use drivers to test components

- To simplify error localisation, systems should be incrementally integrated.

Software Testing Process

SYSTEM TESTING

System Testing

- Testing of groups of components integrated to create a system or sub-system;

- The responsibility of an independent testing team;

- Tests are based on a system specification.

System Testing

- Functional testing
  - Test end to end functionality
  - Requirement focus
    - Test cases derived from specification
  - Use-case focus
    - Test selection based on user profile

System Testing

- 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.

Software Testing Process

ACCEPTANCE TESTING

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

Software Testing Process

REGRESSION TESTING

Re-Test and Regression Testing

- 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
  - 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
  - Should preferably be automated

Software Testing Process

TEST STRATEGIES

Strategies

- Code coverage strategies, e.g.
  - Decision coverage
  - Path coverage
  - Data-Flow analysis (Defines -> Uses)
- Specification-based testing, e.g.
  - Equivalence partitioning
  - Boundary-value analysis
  - Combination strategies
- State-based testing

Test Strategies

- Black-box or behavioral testing
  - knowing the specified function a product is to perform and demonstrating correct operation based solely on its specification without regard for its internal logic
-
  - knowing the internal workings of a product, tests are performed to check the workings of all possible logic paths

Code Coverage

- Statement coverage
  - Each statement should be executed by at least one test case
  - Minimum requirement

- Branch/Decision coverage
  - All paths should be executed by at least one test case
  - All decisions with true and false value

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

Mutation Testing

- Mutants

```
int getMax(int x, int y) {
    int max;

    if (x >y)
        max = x;
    else
        max = y;
    return max;
}
```

```
int getMax(int x, int y) {
    int max;

    if (x >=y)
        max = x;
    else
        max = y;
    return max;
}
```

```
int getMax(int x, int y) {
    int max;

    if (x >y)
        max = x;

    else
        max = x;
return max;
}
```

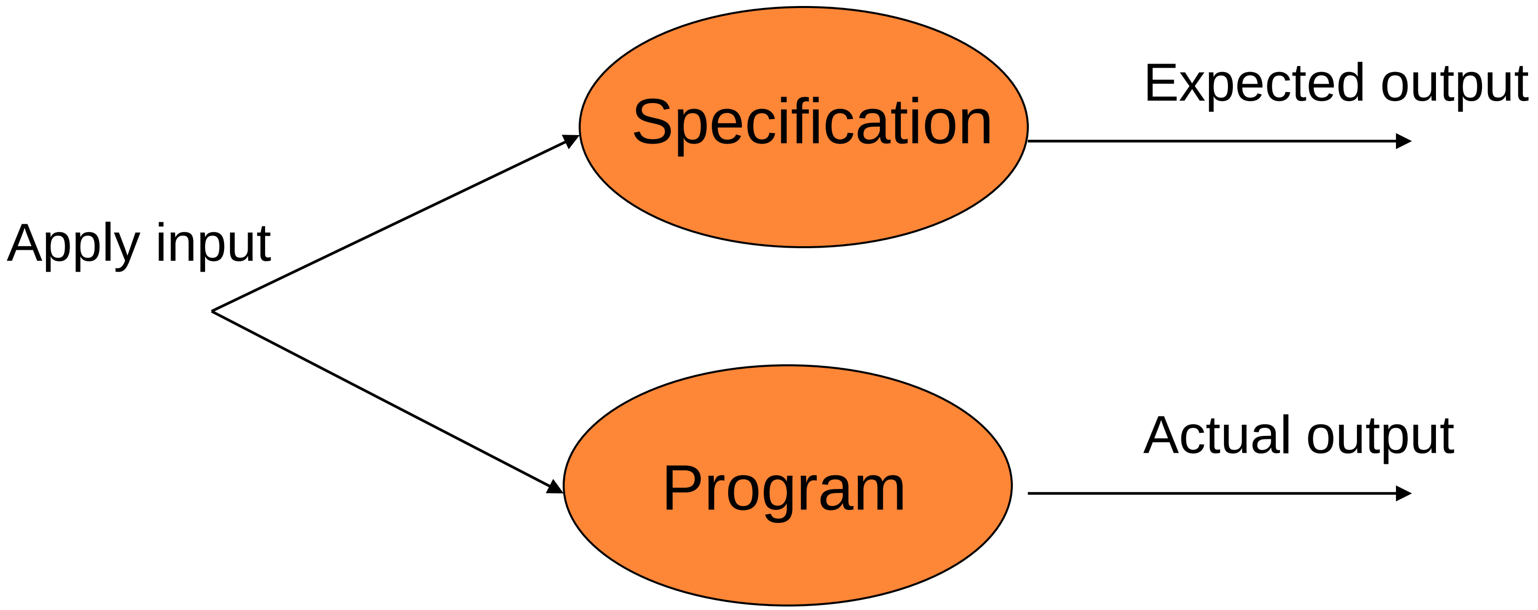
Examples of Mutant Operators

Mutant operator	In program	In mutant
Variable replacement	z=x*y+1;	x=x*y+1; z=x*x+1;
Relational operator replacement	if (x<y)	if(x>y) if(x<=y)
Off-by-1	z=x*y+1;	z=x*(y+1)+1; z=(x+1)*y+1;
Replacement by 0	z=x*y+1;	z=0*y+1; z=0;
Arithmetic operator replacement	z=x*y+1;	z=x*y-1; z=x+y-1;

Specification-based testing

- Test cases derived from specification
- Equivalence partitioning
  - Identify sets of input from specification
    - Assumption: if one input from set *s* leads to a failure, then all inputs from set *s* will lead to the same failure
  - Chose a representative value from each set
  - Form test cases with the chosen values

Specification-Based Testing



Validate the observed output against the expected output

Black Box testing
Equivalence Partitioning

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

Equivalence Partitioning

- Black-box technique divides the input domain into classes of data from which test cases can be derived.

- An ideal test case uncovers a class of errors that might require many arbitrary test cases to be executed before a general error is observed.

Specification-based testing

- Boundary value analysis
  - Identify boundaries in input and output
  - For each boundary:
    - Select one value from each side of boundary (as close as possible)
  - Form test cases with the chosen values

Boundary Value Analysis

- Black-box technique
  - focuses on classes and also on the boundaries of the input domain.

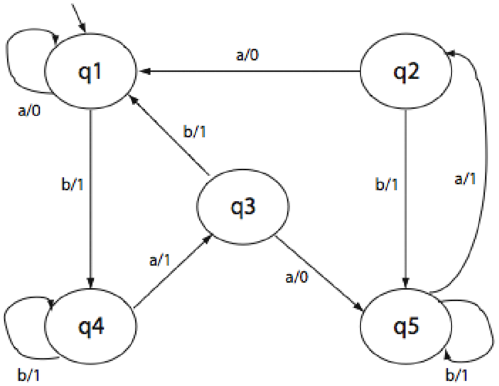
- Guidelines:

1. If input condition specifies a range bounded by values a and b, test cases should include a and b, values just above and just below a and b

2. If an input condition specifies a number of values, test cases should exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values

STATE-BASED TESTING

- Model functional behavior in a state machine (communication – protocol …)
- Select test cases in order to cover the graph
  - Each node
  - Each transition
  - Each pair of transitions
  - Each chain of transitions of length *n*



Example:
Factorial function: n!

Factorial of n: n!

- Equivalence partitioning – break the input domain into different classes:
  - Class1: $n<0$
  - Class2: $n>0$ and n! doesn't cause an overflow
  - Class3: $n>0$ and n! causes an overflow
- Boundary Value Analysis:
  - $n=0$ (between class1 and class2)

Factorial of n: n!
Test cases

- Test case = (ins, expected outs)
- Equivalence partitioning – break the input domain into different classes:
  1. From Class1: ((n = −1), " function not defined for n negative")
  2. From Class2: ((n = 3), 6)
  3. From Class3: ((n=100), " input value too big")
- Boundary Value Analysis:
  4. ((n=0), 1)

White-Box Testing

Structural Testing

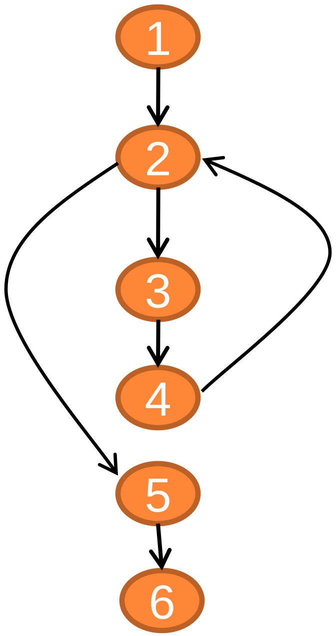
Test Strategies

White Box testing
Structural Testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
- Statements with conditions are therefore nodes in the flow graph.

Path testing – Control Flow Graph

- White-box technique is based on the **program flow graph (CFG)**



Many paths between 1 (begin) and 6 (end)

1, 2, 5, 6

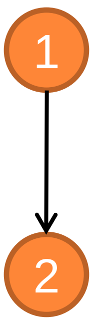
1, 2, 3, 4, 2, 6

1, 2, 3, 4, 2, 3, 4, 2, 5, 6

…

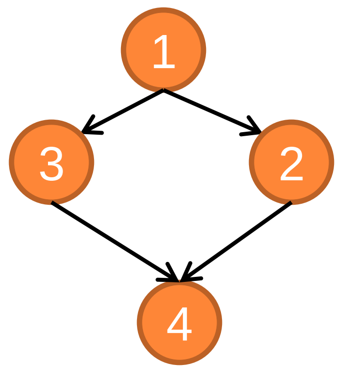
- Prepare test cases that will force the execution of each path in the basis set.
  - Test case : ((inputs …) , (expected outputs …))

Program flow graph
Basic Control Flow Graphs



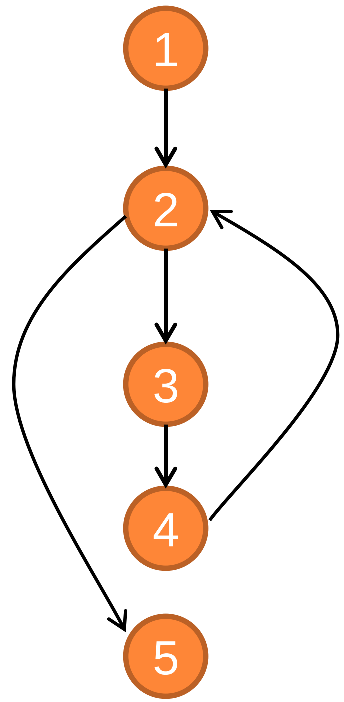
A sequence:

X = 1;
Y = X * 10;

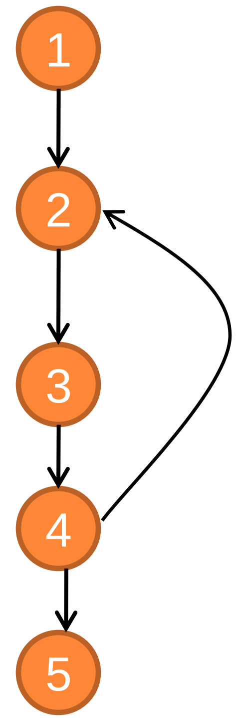


If condition:

If … Then
…
Else
…
End if



While loop:
While … do
…
statements
…
End while



Do While loop (Repeat until):
do
…
statements
…
While …

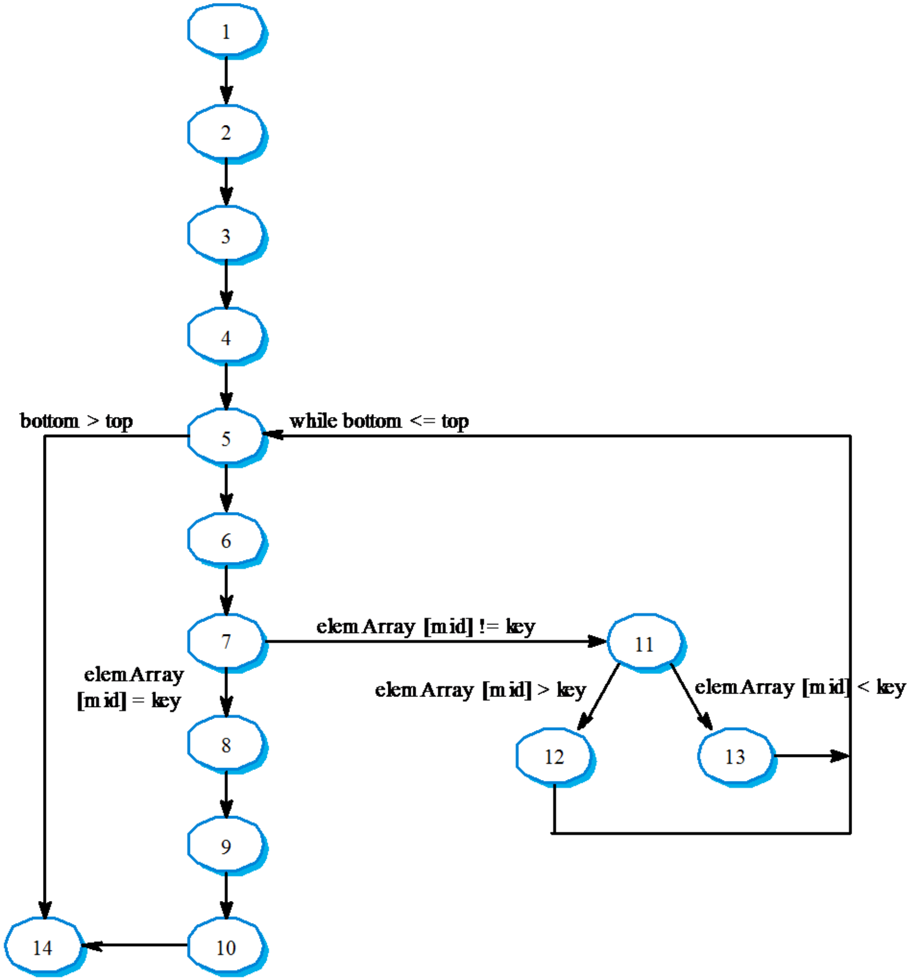
Example:
Binary search pseudo-code

```
class BinSearch {

// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types
by
// reference to a function and so return two values
// the key is -1 if the element is not found

        public static void search ( int key, int [] elemArray, Result r )
        {
1.              int bottom = 0 ;
2.              int top = elemArray.length - 1 ;
                int mid ;
3.              r.found = false ;
4.              r.index = -1 ;
5.              while ( bottom <= top )
                {
6                       mid = (top + bottom) / 2 ;
7                       if (elemArray [mid] == key)
                        {
8                               r index = mid ;
9                               r.found = true ;
10                               return ;
                        } // if part
                        else
                        {
11                              if (elemArray [mid] < key)
12                                      bottom = mid + 1 ;
                                else
13                                      top = mid - 1 ;
                         }
                } //while loop
14.     } // search
} //BinSearch
```

Binary search flow graph



Independent Paths

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
- 1, 2, 3, 4, 5, 14
- 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, …
- 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …

- Test cases should be derived so that all of these paths are executed

Software Metrics

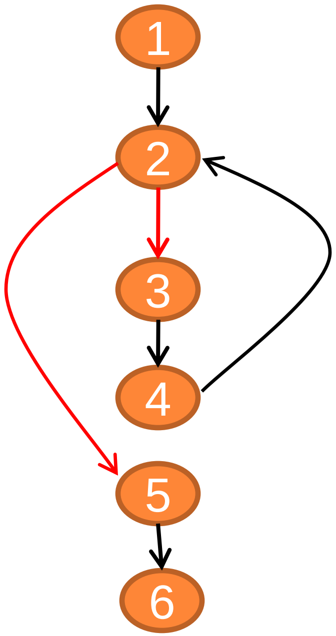
- McCabe's cyclomatic number, introduced in 1976, is, after lines of code, one of the most commonly used metrics in software development.
- The cyclomatic complexity of the program is computed from its control flow graph (CFG) using the formula:

$$V(G) = Edges - Nodes + 2$$

or by counting the conditional statements and adding 1

- This measure determines the basis set of linearly independent paths and tries to measure the complexity of a program

Software Metrics



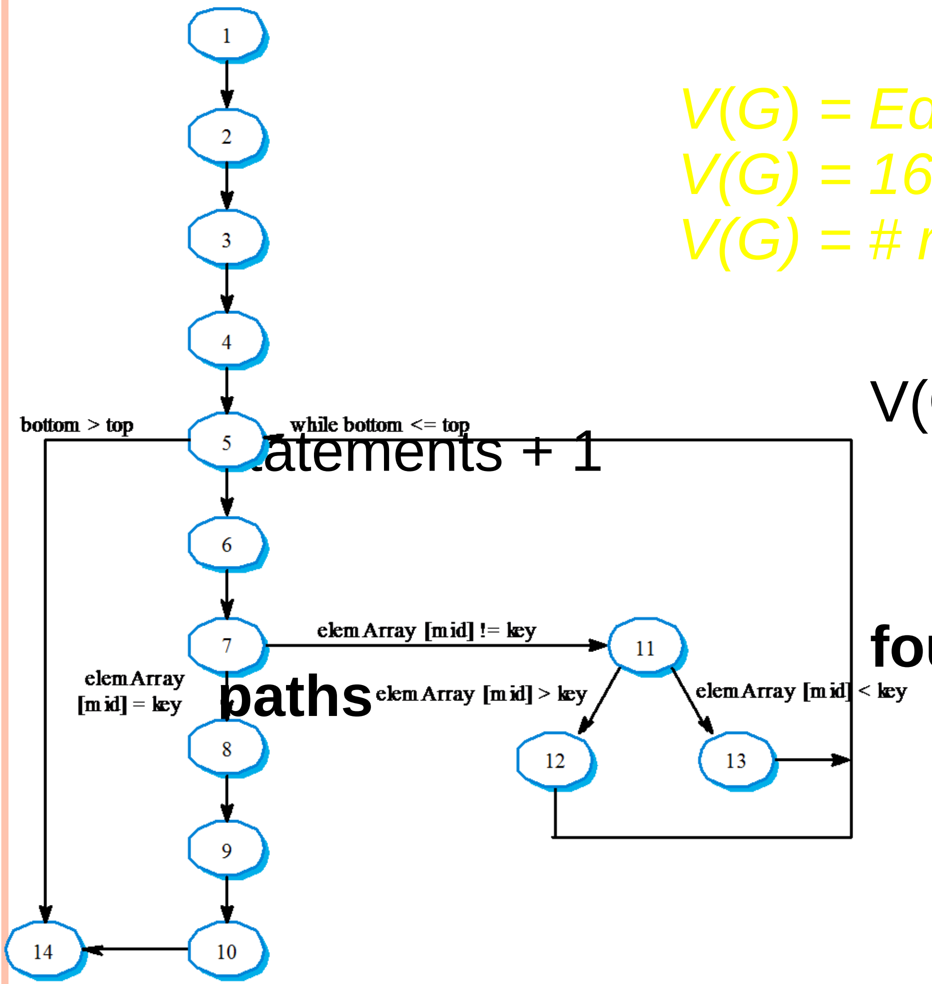
V(G) = Edges – Nodes + 2
*V(G) = 6 – 6 + 2 = **2***

V(G) = conditional statements + 1
= 1 + 1 = **2**

Two linearly independent paths:

1, 2, 5, 6
1, 2, 3, 4, 2, 5, 6

SOFTWARE METRICS



V(G) = Edges – Nodes + 2
*V(G) = 16 – 14 + 2 = **4***
V(G) = # regions = 4

V(G) = conditional statements + 1

= 3 + 1 = **4**

four linearly independent paths

References

- K. Naik and P. Tripathy: *"Software Testing and Quality Assurance"*, Wiley, 2008.
- *Ian Sommerville, Software Engineering, 8th edition, 2006.*
- *Aditya P. Mathur,"Foundations of Software Testing", Pearson Education, 2009.*
- *D. Galin, "Software Quality Assurance: From Theory to Implementation", Pearson Education, 2004*
- *David Gustafson, "Theory and Problems of Software Engineering",* Schaum's Outline Series*,* McGRAW-HILL, *2002.*