Chapter 1

Software Testing

Contents

What is testing?

- Several definitions:
  - "Testing is the process of establishing confidence that a program or system does what it is supposed to." (Hetzel 1973)
  - "Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results." (Hetzel 1983)
  - "Testing is the process of executing a program or system with the intent of finding errors." (Myers 1979)
  - The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component (IEEE)
  - Testing is not
    - the process of demonstrating that errors are not present.

Background

- Main objectives of a software project is to be productive and producing high quality software products
  - Quality has many dimensions: reliability, maintainability, portability etc.
- Reliability is perhaps the most important
  - Reliability reefers the chances of software failing
- More defects implies more chances of failure, which in turn means lesser reliability
  - Hence to develop high quality software, minimize bugs/errors as much as possible in the delivered software
- Generally, testing involves
  - Demonstrating the system customer that the software meets its requirements; (Validation testing)
  - Discovering faults or defects in the software where its behavior is incorrect or not in conformance with its specification (Defect testing)

Background…

- Testing only reveals the presence of defects
  - It doesn't identify nature and location of defects
- Identifying & removing the defect is role of debugging and rework
- Testing is expensive
  - Preparing test cases, performing testing, defects identification & removal all consume effort
    - Overall testing becomes very expensive : 30-50% development cost
- Who is involved in testing?
  - Software Test Engineers and Testers
  - Test manager
  - Development Engineers
  - Quality Assurance Group and Engineers

Software Testing: Terminologies

- **Error, Mistake, Bug, Fault and Failure**
  - **Error** is a mistake made by an engineer .
    - This may be a syntax error, misunderstanding of specifications or logical errors.
    - **Bugs** are coding mistakes/errors.
  - A **fault/defect** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc.
  - A **failure** is an incorrect output/behavior that is caused by executing a fault.
    - A particular fault may cause different failures, depending on how it has been exercised.

Software Testing: Terminologies…

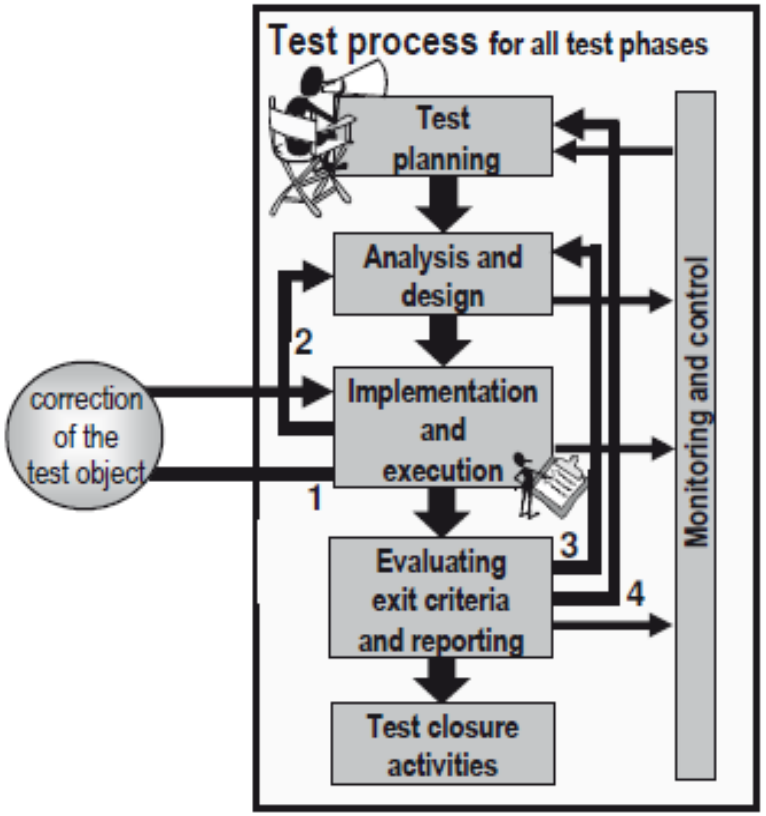
- **Test, Test Case and Test Suite**
  - **Test cases** are inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification
    - During testing, a program is executed with a set of test cases
      - failure during testing shows presence of defects.
  - **Test/Test Suite**: A set of one or more test cases
- **Verification and Validation**
  - Verification : the software should conform to its specification.
    - i.e. Are we building the product right".
  - Validation: The software should do what the user really requires.
    - i.e."Are we building the right product".
      - Testing= Verification +Validation

Testing and the life cycle

- Testing is available in every software development phases
- Requirements engineering
  - Requirements should be tested for completeness, consistency, feasibility, testability, … through reviews & the like
  - Typical errors that could discovered are missing, wrong, extra information, …
- Design
  - The design itself can be tested using review or formal verification techniques if the design conforms with the requirements
- Implementation
  - Check consistency of implementation and previous documents using all kinds of functional and structural test techniques
- Maintenance
  - Regression testing: either retest all, or a more selective retest

Software Testing Life Cycle (STLC)

- Software testing has its own life cycle that intersects with every stage of the SDLC.
- It identifies what test activities to carry out and when (what is the best time) to accomplish those test activities.
- Even though testing differs between organizations, STLC consists of the following (generic) phases:
  - Test Planning
  - Test Analysis
  - Test Design
  - Construction and verification
  - Testing Execution
  - Final Testing
  - Post Implementation.



Software Testing Life cycle		
Phase	**Activities**	**Outcome**
Planning	Create high level test plan	Test plan, Refined Specification
Analysis	Create detailed test plan, Functional Validation Matrix, test cases	Revised Test Plan, Functional validation matrix, test cases
Design	test cases are revised; select which test cases to automate	revised test cases, test data sets, sets, risk assessment sheet
Construction	scripting of test cases to automate,	test procedures/Scripts, Drivers, test results, Bugreports.
Testing cycles	complete testing cycles	Test results, Bug Reports
Final testing	execute remaining stress and performance tests, complete documentation	Test results and different metrics on test efforts
Post implementation	Evaluate testing processes	Plan for improvement of testing process

Test case design

- Test case design involves designing the test cases (inputs and outputs) used to test the system.
- The goal of test case design is to create a set of tests that are effective in validation and defect testing.
- Two approaches to design test cases are
  - Functional/ behavioral/ black box testing
  - Structural or white box testing
- Spending sufficient time in test case design helps to get “good” test cases.
- Both are complimentary; we discuss a few approaches/criteria for both

Black Box testing

- In black box testing items are treated as black whose logic is unknown
- All that is known is what's goes in and what comes out. the input and the output
- Focuses on the functional requirement of the software also called behavioral testing
- Does not need any knowledge of internal design or code
- Tester is needed to be thorough with the requirement specifications of the system.
- User should know how the system should behave in response to the particular action

Black Box testing

- Black-box testing attempts to find errors in the following categories:
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external data base access.
  - Behavior or performance errors
  - Initialization and termination errors.

Black Box testing...

- **Advantages**
  - Tester can be non-technical.
  - Test cases can be designed as soon as the functional specifications are complete
- **Disadvantages**
  - The tester can never be sure of how much of the system under test has been tested.
    - i.e. chances of having unidentified paths during this testing
  - The test inputs needs to be from large sample space.

Black box testing approaches

- Requirements-based testing
- Equivalence Class partitioning
- Boundary value analysis
- Cause Effect graphing

Requirements-based testing

- A general principle of requirements engineering is that requirements should be testable.
- Requirements-based testing is a validation testing technique where you consider each requirement and derive a set of tests for that requirement.
- **Significance**
  - **Alignment with Design Specifications**
  - **Verification of Documented Requirements**
  - **Systematic Feature Evaluation**
  - **Inconsistency Identification**
  - **Improvement in the reliability and overall quality**
- *Testing Effectiveness = (Number of Test Cases Validated Against Requirements / Total Number of Test Cases) × 100*

Requirements-based testing …

Example: E-commerce website login functionality

Requirement:

- The system must allow a registered user to log in with a valid username and password.
- If incorrect credentials are entered, an error message should be displayed.
- The login button should remain disabled until both the username and password fields are filled.

- **Test Case 1: Successful login**
- **Input**: Enter valid username and password, then click "Login."
- **Expected Result**: User is logged in successfully and redirected to the homepage.

Test Case 2: Invalid username or password

- **Input**: Enter an invalid username and/or password, then click "Login."
- **Expected Result**: The system displays the error message, *"Invalid username or password."*

Test Case 3: Empty fields

- **Input**: Leave the username or password field empty.
- **Expected Result**: The "Login" button remains disabled.

Test Case 4: Case sensitivity

- **Input**: Enter a valid username but with incorrect capitalization and correct password.
- **Expected Result**: The system displays the error message, *"Invalid username or password."*

Test Case 5: Special characters in username or password

- **Input**: Enter special characters in either username or password fields.
- **Expected Result**: The system handles input gracefully, showing appropriate validation messages or error prompts as required.

Equivalence Class partitioning

- Divide the input space into equivalent classes
- If the software works for a test case from a class then it is likely to work for all
- Can reduce the set of test cases if such equivalent classes can be identified
- Getting ideal equivalent classes is impossible, without looking at the internal structure of the program
- For robustness, include equivalent classes for invalid inputs also
- Example: Look at the following taxation table

Income	Tax Percentage
Up to and including 500	0
More than 500, but less than 1,300	30
1,300 or more, but less than 5,000	40

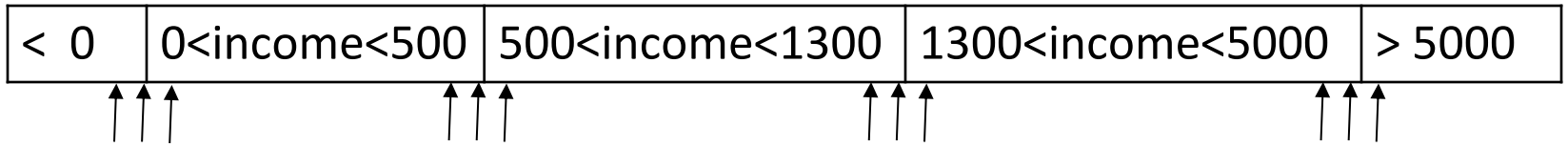
Equivalence Class partitioning…

- Based on the above table 3 valid and 4 invalid equivalent classes can be found
  - Valid Equivalent Classes
    - Values between 0 to 500, 500 to 1300 and 1000 to 5000
  - Invalid Equivalent Classes
    - Values less than 0, greater than 5000, no input at all and inputs containing letters
- From this classes we can generate the following test cases

Test Case ID	Income	Tax
1	200	0
2	1000	300
3	3500	1400
4	-4500	Income can’t be negative
5	6000	Tax rate not defined
6		Please enter income
7	98ty	Invalid income

Boundary value analysis

- It has been observed that programs that work correctly for a set of values in an equivalence class fail on some special values.
- These values often lie on the boundary of the equivalence class.
- A boundary value test case is a set of input data that lies on the edge of a equivalence class of input/output
- Example
  - Using an example in ECP generate test cases that provides 100% BVA coverage.



  - SO, we need from12 – 14 (2 for no and character entries) test cases to achieve the aforementioned coverage

Cause Effect graphing

- Equivalence classes and boundary value analysis consider each input separately
- To handle multiple inputs, different combinations of equivalent classes of inputs can be tried
- Number of combinations can be large – if n is different input conditions such that each condition is valid/invalid, then we will have 2^n test cases.
- Cause effect graphing helps in selecting combinations as input conditions
  - A *cause* is a distinct input condition, and an *effect* is a distinct output condition (T/F).

Cause Effect graphing…

- The steps in developing test cases with a cause-and-effect graph are as follows
  - Decompose the specification of a complex software component into lower-level units.
  - Identify causes and their effects for each specification unit.
  - Define the rules
  - Create Boolean cause-and-effect graph.
    - Causes and effects are nodes
    - dependency b/n nodes are represented by arcs
  - Annotate the graph with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints (if needed).
  - The graph is then converted to a decision table.
  - The columns in the decision table are transformed into test cases

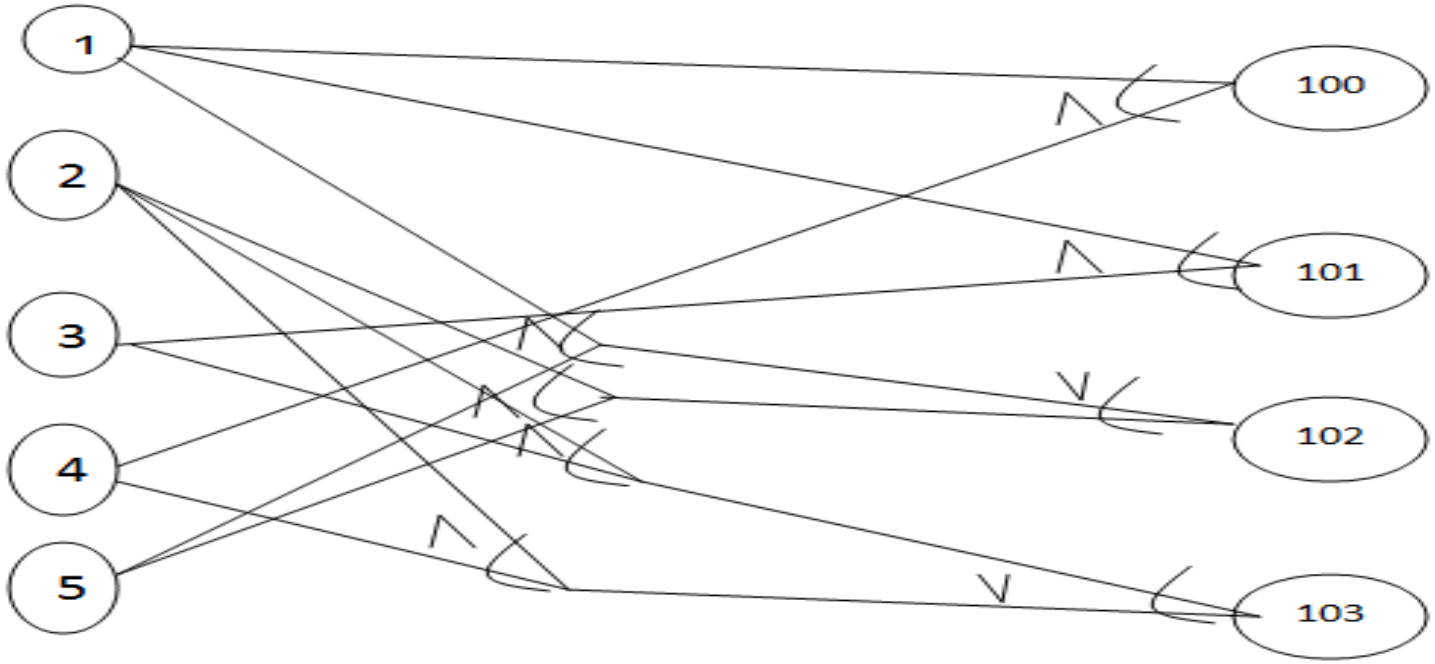
Cause Effect graphing…

- Consider the following set of requirements as an example:
  - Requirements for Calculating Car Insurance Premiums:
    - R00101 For females less than 65 years of age, the premium is $500
    - R00102 For males less than 25 years of age, the premium is $3000
    - R00103 For males between 25 and 64 years of age, the premium is $1000
    - R00104 For anyone 65 years of age or more, the premium is $1500
- The causes and their effects identified from the specification are

Causes (input conditions)	Effects (output conditions)
1. Sex is Male	100. Premium is $1000
2. Sex is Female	101. Premium is $3000
3. Age is <25	102. Premium is $1500
4. Age is >=25 and < 65	103. Premium is $500
5. Age is >= 65	

Cause Effect graphing…

- Define the Rules
  - If sex is female and age is less than 65, the premium is $500
  - If sex is male and age is less than 25, the premium is $3000
  - If sex is male and age is b/n 25 and 65, the premium is $1000
  - If sex is male or female and age is greater than 65, the premium is $1500
- Now, Draw the cause/ Effect graph from the rules



Cause Effect graphing…

- The graph is converted to decision table
  - Since we have 5 input conditions we will have 25 = 32 columns
  - However , for simplicity only 6 are shown in this *limited-entry decision table.*

Test Case	1	2	3	4	5	6
Causes:						
1 (male)	1	1	1	0	0	0
2 (female)	0	0	0	1	1	1
3 (<25)	1	0	0	0	1	0
4 (>=25 and < 65)	0	1	0	0	0	1
5 (>= 65)	0	0	1	1	0	0
Effects:						
100 (Premium is $1000)	0	1	0	0	0	0
101 (Premium is $3000)	1	0	0	0	0	0
102 (Premium is $1500)	0	0	1	1	0	0
103 (Premium is $500)	0	0	0	0	1	1

  - From this table the test cases can be generated as shown in the next table

Cause Effect graphing…

- Generate the test cases
  - The columns in the decision table are converted into test cases.

Test Case #	Inputs (Causes)		Expected Output (Effects)
	Sex	Age	Premium
1	Male	<25	$3000
2	Male	>=25 and < 65	$1000
3	Male	>= 65	$1500
4	Female	>= 65	$1500
5	Female	<25	$500
6	Female	>=25 and < 65	$500

White box testing

- White-box testing(also called clear box testing, glass box testing, transparent box testing, or structural testing)is a method of testing software that tests internal structures or workings of an application[the tester has access to the internal data structures and algorithms including the code that implement these.]

Using the White Box Approach to Test Case Design:

White box testing methods are especially useful for revealing design and code-based control, logic and sequence defects, initialization defects, and data flow defects.

Control flow based criteria

- Considers the program as control flow graph - Nodes represent code blocks – i.e. set of statements always executed together
  - An edge (i, j) represents a possible transfer of control from node *i* to node *j*.
- Any control flow graph has a start node and an end node
- A *complete path* (or a path) is a path whose first node is the start node and the last node is an exit node.
- Control flow graph has a number of coverage criteria. These are
  - Statement Coverage Criterion
  - Branch coverage
  - Linearly Independent paths
  - (ALL) Path coverage criterion

Statement Coverage Criterion

- The simplest coverage criteria is statement coverage;
  - Which requires that each statement of the program be executed at least once during testing.
  - I.e. set of paths executed during testing should include all nodes
- This coverage criterion is not very strong, and can leave errors undetected.
  - Because it has a limitation in that it does not require a decision to evaluate to false if no else clause
  - E.g. : abs (x) : if (x>=0) x = -x; return(x)
    - The set of test cases {x = 0} achieves 100% statement coverage, but error not detected
  - Guaranteeing 100% coverage not always possible due to possibility of unreachable nodes

Cont...

- *Statement coverage = (Number of executed statements / Total number of statements in source code) * 100*
- *number of test cases for 100% statement coverage = 1 + number of else statement(nested if)*
- *number of test cases for 100% statement coverage = 2(with else unnested)*
- *number of test cases for 100% statement coverage = 1(with no else unnested)*

```
Read A
Read B
if A > B
    Print “A is greater than B”
else
    Print “B is greater than A”
endif
```

example

Case 1:

If A = 7, B= 3

No of statements Executed= 5

Total statements= 7

Statement coverage= 5 / 7 * 100

= 71.00 %

Case 2:

If A = 4, B= 8

No of statements Executed= 6

Total statements= 7

Statement coverage= 6 / 7 * 100

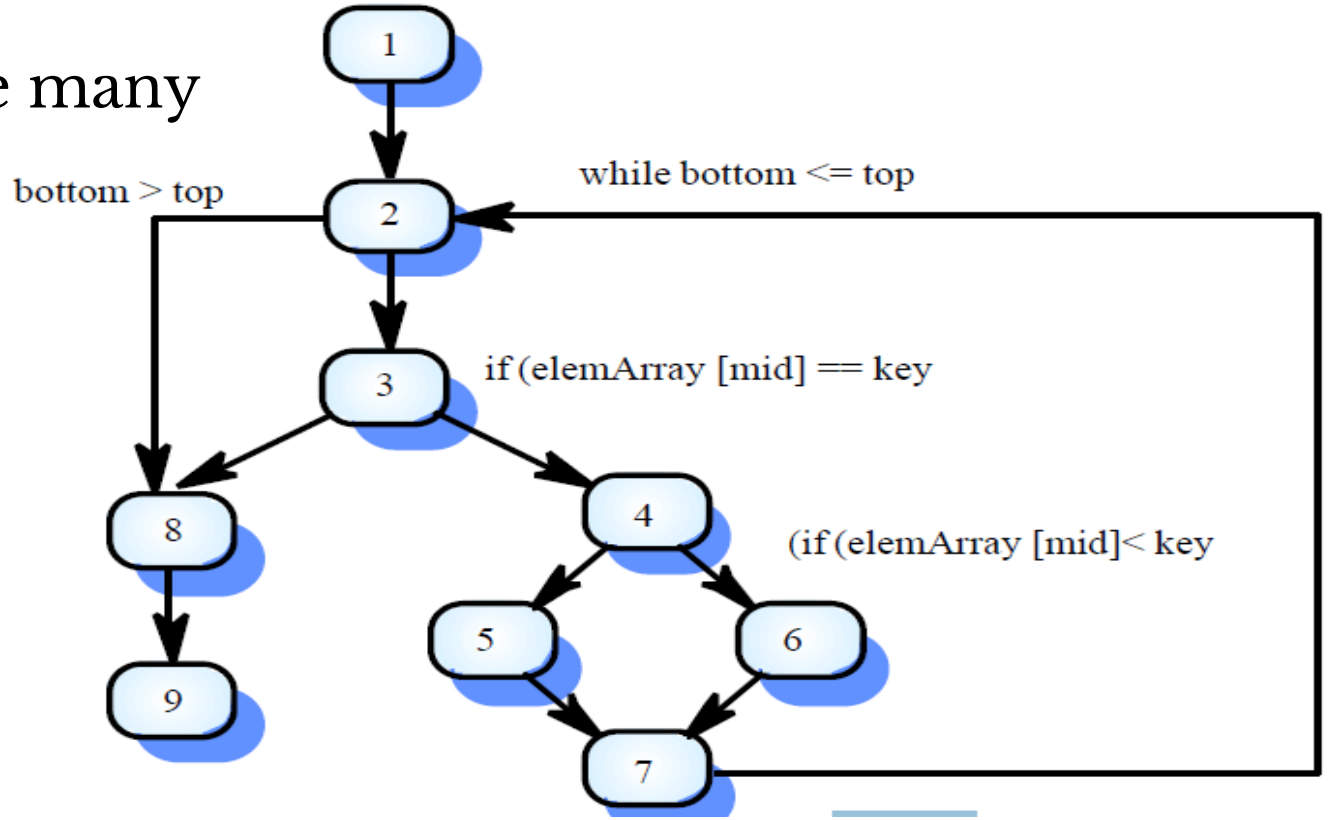
= 85.20 %

Branch coverage

- A little more general coverage criterion is branch coverage
  - which requires that each edge in the control flow graph be traversed at least once during testing.
  - i.e. branch coverage requires that each decision in the program be evaluated to true and false values at least once during testing.
- Branch coverage implies statement coverage, as each statement is a part of some branch.
- The trouble with branch coverage comes if a decision has many conditions in it (consisting of a Boolean expression with Boolean operators *and* & *or*).
- In such situations, a decision can evaluate to true and false without actually exercising all the conditions.
- This problem can be resolved by requiring that all conditions evaluate to true and false (Condition Coverage)

Linearly Independent paths

- modeling paths through an application and then using tests to exercise those paths.
- What we've seen so far is that while many path segments are covered repeatedly, others may only be touched once. Prepare test cases that covers all linearly independent Paths



- Linearly independent paths are (1, 2, 8, 9), (1, 2, 3, 8, 9), (1, 2, 3, 4, 5, 7, 2 , 8, 9), and (1, 2, 3, 4, 6, 7, 2, 8, 9)
- Test cases should be derived so that all of these paths are executed

(ALL) Path coverage criterion

- 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.
- The difficulty with this criterion is that programs that contain loops can have an infinite number of possible paths.

Example

- The following piece of C code takes integer inputs x and y and computes x^y. Answer the subsequent questions based on the code.

```
1. scanf(x, y); if (y < 0)
2.       pow = 0 - y;
3. else pow = y;
4.       z = 1.0;
5. while (pow != 0)
6.       {z = z * x; pow = pow - 1; }
7. if (y < 0)
8.       z = 1.0 / z;
9. printf(z);
```

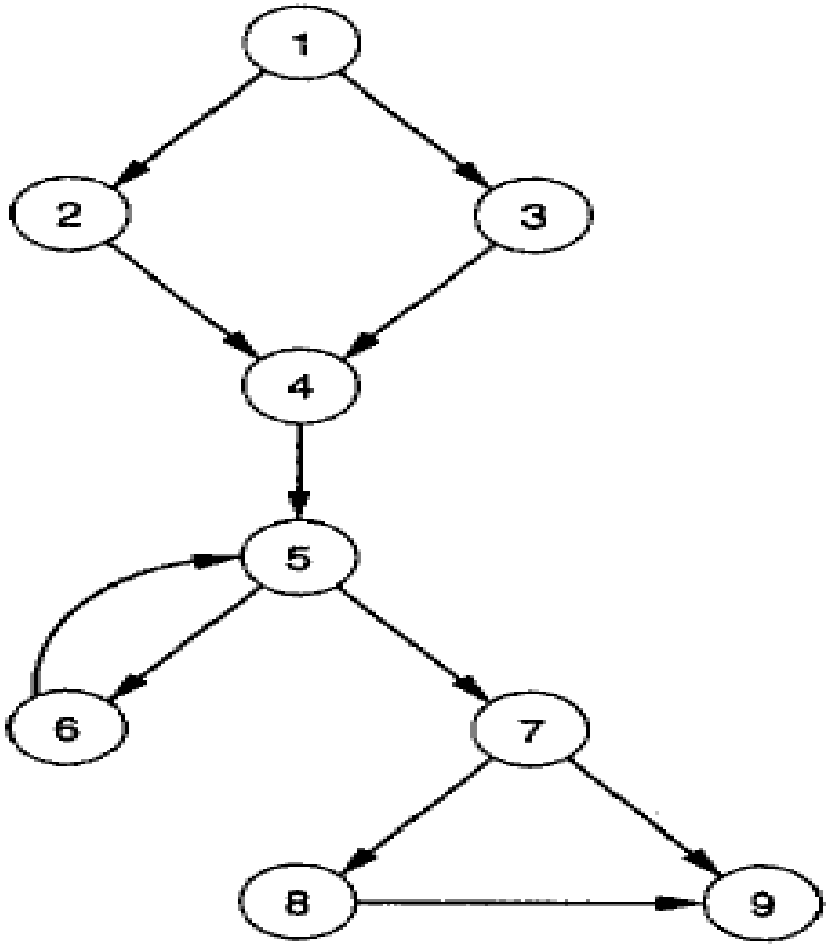
- For the code, do the following
  1. Draw control flow graph
  2. Select a set of test cases that will provide
     A. 100% statement coverage.
     B. 100% branch coverage.
     C. 100% linearly independent path coverage.
     D. 100% path coverage.

Example...

Answers

1. Data flow diagram

The following test cases enables to achieve 100% statement coverage



X	Y	X^Y	Nodes / Statements Covered
2	2	4	1,3,4,5,6,7,9
2	-2	0.25	1,2,4,5,6,7,8,9

Example...

- The same test cases as statement coverage enables 100% branch coverage too

X	Y	X^Y	Branches Covered
2	2	4	1→3→4→5→6→5→6→5→7→9
2	-2	0.25	1→2→4→5→6→5→6→5→7→8→9

- There are four linearly independent paths. The following test cases are required

X	Y	X^Y	Paths Covered
Infeasible			1→2→4→5→7→9
Infeasible			1→2→4→5→7→8→9
2	-1	0.5	1→2→4→5→6→5→7→8→9
2	0	1	1→3→4→5→7→8→9

- There are infinite paths for the program. So, achieving 100% all path coverage is impossible.

Exercise

- Suppose three numbers A, B, and C are given in ascending order representing the lengths of the sides of a triangle. The problem is to determine the type of the triangle (whether it is isosceles, equilateral, right, obtuse, or acute).
- Answer all questions of an example

```
1.  read(a, b, c);
2.  if (a < b) or (b < c) then
3.          print ("Illegal inputs");
4.          return;
5.  if (a=b) or (b=c) then
6.     if (a=b) and (b=c) then
7.          print ("equilateral triangle");
8.     else print ("isosceles triangle");
9.  else
10.             a := a*a; b := b*b; c := c*c;
11.             d := b+c;
12.             if (a = d) then
13.                     print ("right triangle");
14.             else if (a<d) then
15.                     print("acute triangle");
16.             else print ("obtuse triangle");
17.         end; //of the code
```

Data flow-based testing

- There are many criteria in control flow based testing including – statement coverage, branch coverage, path...
- None is sufficient to detect all types of defects (e.g. a program missing some paths cannot be detected)
- Data flow-based testing involves selecting the paths to be executed during testing based on data flow analysis (dynamic testing), rather than control flow analysis (static testing).
- A statement in the control flow graph (in which each statement is a node) can be used for this type of testing because it can be easily converted to def/use graph.
- A def-use graph is constructed by associating variables with nodes and edges in the control flow graph
- Once the graph have been drawn different data flow based criteria can be applied for test case generation.

Mutation Testing

- Mutation testing takes the program and creates many mutants of it by making simple changes to the program (for example introducing faults)
- The goal of this testing is to make sure that during the course of testing, each mutant produces an output different from the output of the original program.
- Clearly this technique will be successful only if the changes introduced in the main program capture the most likely faults in some form.
  - If a program P contains an expression *a =b * (c-d)*, how many mutants will be produced by replacing the arithmetic operator by the following operators. {*, +, -, /}

Chapter 1: Software Testing ...

Part two

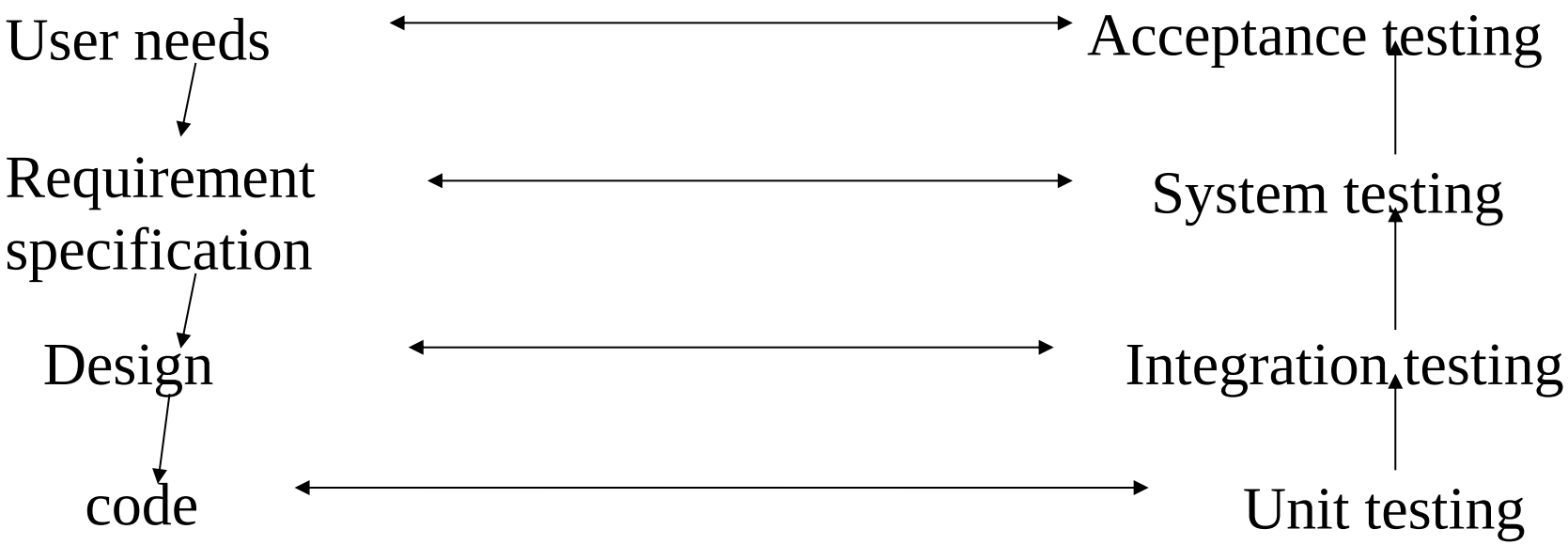
Contents

- Testing Process
- Levels of Testing
  - Unit, Integration, System and Acceptance Testing
- Other Forms of Testing
- Static Testing
- Test Plan
- Test case specifications, execution and Analysis
- Test automation
- Limitations of Testing
- Debugging
- Software Quality Assurance

Levels of Testing

- Execution-based software testing, especially for large systems, is usually carried out at different levels.
- In most cases there will be 3–4 levels, or major phases of testing: unit test, integration test, system test, and some type of acceptance test
- The code contains requirement defects, design defects, and coding defects
- Nature of defects is different for different injection stages
- One type of testing will be unable to detect the different types of defects
  - different levels of testing are used to uncover these defects

Levels of Testing



- The major testing levels are similar for both object-oriented and procedural-based software systems.
- Basically the levels differ in
  - the element to be tested
  - responsible individual
  - testing goals

Different Levels of Testing…

- **Unit Testing**
  - ***Element to be tested :*** individual component (method, class or subsystem)
  - ***Responsible individual:*** Carried out by developers
  - ***Goal:*** Confirm that the component or subsystem is correctly coded and carries out the intended functionality
    - Focuses on defects injected during coding: coding phase sometimes called “coding and unit testing”
- **Integration Testing**
  - ***Element to be tested :*** Groups of subsystems (collection of subsystems) and eventually the entire system
  - ***Responsible individual:*** Carried out by developers
  - ***Goal:*** Test the interfaces among the subsystems.
    - i.e. for problems that arise from component interactions.

Different Levels of Testing…

- **System Testing**
  - ***Element to be tested :*** The entire system
  - ***Responsible individual:*** Carried out by separate test team
  - ***Goal:*** Determine if the system meets the requirements (functional and nonfunctional)
  - Most time consuming test phase
- **Acceptance Testing**
  - ***Element to be tested :*** Evaluates the system delivered by developers
  - ***Responsible individual:*** Carried out by the client. May involve executing typical transactions on site on a trial basis
  - ***Goal:*** Demonstrate that the system meets/satisfies user needs?

  - Only after successful acceptance testing the software deployed

Different Levels of Testing…

- If the software has been developed for the mass market (shrink wrapped software), then testing it for individual users is not practical or even possible in most cases.
- Very often this type of software undergoes two stages of acceptance test: Alpha and Beta testing
- *Alpha test*.
  - This test takes place at the developer's site.
  - Testing done using simulated data in a lab setting
  - Developers observe the users and note problems.
- *Beta testing*
  - the software is sent to a cross-section of users who install it and use it under real world working conditions with real data.
  - The users send records of problems with the software to the development organization

Different Levels of Testing…

- The levels discussed earlier are performed when a system is being built from the components that have been coded
- Another level of testing, called *regression testing*, is performed when some changes are made to an existing system.
  - Regression testing usually refers to testing activities during software maintenance phase.
- **Regression testing**
  - makes sure that the modification has not had introduced new errors
  - ensures that the desired behavior of the old services is maintained
  - Uses some test cases that have been executed on the old system
- Since regression testing is supposed to test all functionality and all previously done changes, regression tests are usually large.
- Thus, regression testing needs automatic execution & checking

Why testing at different levels?

- Implementing all of these levels of testing require a large investment in time and organizational resources.
- However, it has the following advantages
  - Goes with software development phases because software is naturally divided into phases
    - Especially true for some software process model
      - V & W models
  - Makes tracking bugs easy
  - Ensures a working subsystem/ component/ library
  - Makes software reuse more practical

Other Forms of testing

- **Top-down and Bottom-up testing**
  - System is hierarchy of modules - modules coded separately
  - Integration can start from bottom or top
  - Bottom-up requires test ***drivers*** while top-down requires ***stubs***
  - Drivers and stubs are code pieces written only for testing
  - Both may be used, e.g. for user interfaces top-down, for services bottom-up
- **Incremental Testing**
  - Incremental testing involves adding untested parts incrementally to tested portion
  - Increasing testing can catch more defects, but cost also goes up
  - Testing of large systems is always incremental
  - Incremental testing can be done in top-down or bottom fashion

Object Oriented Testing

- Testing begins by evaluating the OOA and OOD models
- OOA models (requirements and use cases) & OOD models (class and sequence diagrams) are tested using
  - Structured walk-throughs, prototypes
  - Formal reviews of correctness, completeness and consistency
- In OO programs the components to be tested are object classes that are instantiated as objects
- Larger grain than individual functions so approaches to white-box testing have to be extended
  - conventional black box methods can be still used
- No obvious ‘top’ to the system for top-down integration and testing
- Object-oriented testing levels
- Testing operations associated with objects
- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system

Static testing

- Static testing is defined as:
  - Testing of a component or system at specification or implementation level without execution of that software (e.g., reviews or static code analysis).“ - (ISEB/ISTQB)
  - In contrast dynamic testing is testing of software where the object under testing, the code, is being executed on a computer.
- Static testing is primarily syntax checking of the code or and manually reading of the code or any document to find errors
- There are a number of different static testing types or techniques
  - Informal reviews
  - Walk-through
  - Technical review
  - Management review
  - Inspection
  - Audit
- The difference b/n these the techniques is depicted in next slide

Static testing...

	Walk-through	Technical review	Management review	Inspection
Primary purpose	Finding defects	Finding defects	Finding defects	Finding defects
Secondary purpose	Sharing knowledge	Make decisions	Monitor and control progress	Process im-provement
Preparation	Usually none	Familiarization	Familiarization	Formal preparation
Usage of basis	Rarely	Maybe	Maybe	Always
Leadership of meeting	Author	As appropriate	As appropriate	Trained moderator
Recommend-ed group size	2–7	3 or more	3 or more	3–6
Formal procedure	Usually not	Sometimes	Sometimes	Always
Volume of material	Relatively low	Moderate to high	Moderate to high	Relatively low
Collection of metrics	Usually not	Sometimes	Sometimes	Always
Output	Sometimes an informal report	More or less formal report	More or less formal report	Defect list, measure-ments, and formal report

Test Plan

- Testing usually starts with test plan and ends with acceptance testing
- Test plan is a general document that defines the scope and approach for testing for the whole project
- Inputs are SRS, project plan, design, code, …
- Test plan identifies what levels of testing will be done, what units will be tested, etc in the project
- It usually contains
  - Test unit specifications: what units need to be tested separately
  - Features to be tested: these may include functionality, performance, usability,…
  - Approach: criteria to be used, when to stop, how to evaluate, etc
  - Test deliverables
  - Schedule and task allocation

IEEE 829 Test Plan Structure

- 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
- Test Deliverables
- Testing Tasks
- Environmental Needs
- Responsibilities
- Staffing And Training Needs
- Schedule
- Risks And Contingencies
- Approvals

Test case specifications

- Test plan focuses on approach; does not deal with details of testing a unit
- Test case specification has to be done separately for each unit
- Based on the plan (approach, features,..) test cases are determined for a unit
- Expected outcome also needs to be specified for each test case
- Together the set of test cases should detect most of the defects
- Set of test cases should be small
  - each test case consumes effort
- Determining a reasonable set of test case is the most challenging task of testing
- The effectiveness and cost of testing depends on the set of test cases
- How to determine if a set of test cases is good?

Test case specifications…

- I.e. the larger set will detect most of the defects, and a smaller set cannot catch these defects
- No easy way to determine goodness; usually the set of test cases is reviewed by experts
- This requires test cases be specified before testing
  - a key reason for having test case specification
- So for each testing, test case specifications are developed, reviewed, and executed
- Test specifications are essentially a table

Seq.No	Condition to be tested	Test Data	Expected result	successful

Test data are Inputs which have been devised to test the system

- Preparing test case specifications is challenging and time consuming

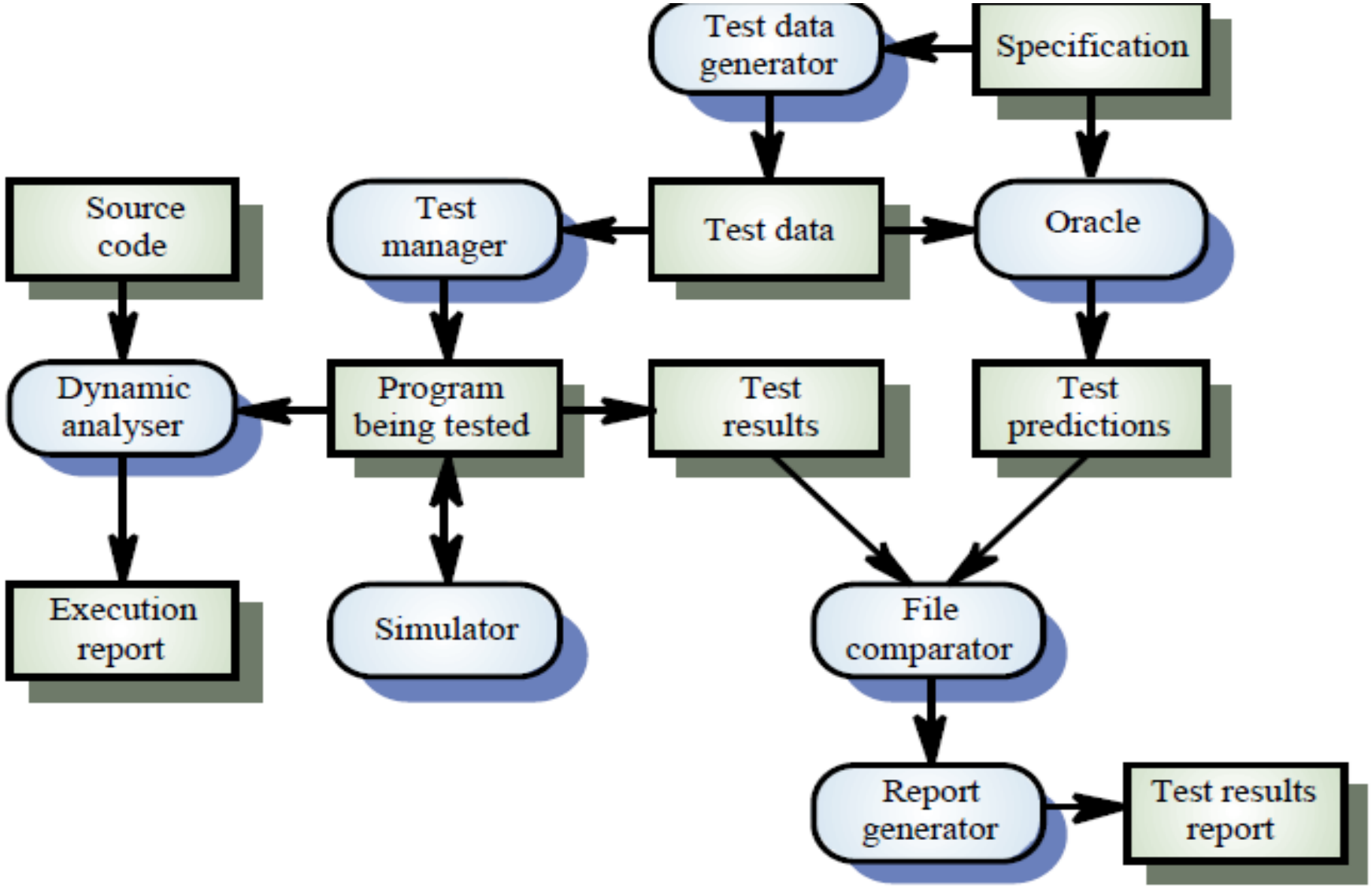
Test case execution and analysis

- Executing test cases may require **drivers** or **stubs** to be written; some tests can be auto, others manual
  - A separate test procedure document may be prepared
  - Test procedure is the steps to be performed to execute the test cases
- Test summary report is often an output
- Test summary gives a summary of test cases executed, effort, defects found, etc
- Monitoring of testing effort is important to ensure that sufficient time is spent
- The error report gives the list of all the defects found.
- To facilitate reporting and tracking of defects found during testing, defects found must be properly recorded and analyzed.

Test automation

- Testing is an expensive process phase.
- Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Systems such as Junit support the automatic execution of tests.
- Most testing workbenches are open systems because testing needs are organisation-specific.
- They are sometimes difficult to integrate with closed design and analysis workbenches.

A testing workbench



Limitations of Testing

- Testing has its own limitations.
  - You cannot test a program completely - Exhaustive testing is impossible
    - You cannot test every path
    - You cannot test every valid input
    - You cannot test every invalid input
  - We can only test against system requirements
    - - May not detect errors in the requirements
    - - Incomplete or ambiguous requirements may lead to inadequate or incorrect testing.
  - Time and budget constraints
    - Compromise between thoroughness and budget.
    - You will run out of time before you run out of test cases
  - Even if you do find the last bug, you'll never know it

Limitations of Testing …

- These limitations require that additional care be taken while performing testing.
- As testing is the costliest activity in software development, it is important that it be done efficiently
  - Test Efficiency – Relative cost of finding a bug in SUT(software under test)
  - Test Effectiveness –ability of testing strategy to find bugs in a software
- Testing should not be done on-the-fly, as is sometimes done.
- It has to be carefully planned and the plan has to be properly executed.
- The testing process focuses on how testing should proceed for a particular project.
  - Various methods of selecting test cases are discussed

Debugging

- Debugging is the process of locating and fixing or bypassing bugs (errors) in computer program code
  - To *debug* a program is to start with a problem, isolate the source of the problem, and then fix it
- Testing does not include efforts associated with tracking down bugs and fixing them.
  - The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979
- Debugging typically happens during three activities in software development:
  - Coding
  - Testing
  - Production/deployment

Types of bugs

- Types of bugs
  - Compile time: syntax, spelling, static type mismatch.
    - Usually caught with compiler
  - Design: flawed algorithm.
    - Incorrect outputs
  - Program logic (if/else, loop termination, select case, etc).
    - Incorrect outputs
  - Memory nonsense: null pointers, array bounds, bad types, leaks.
    - Runtime exceptions
  - Interface errors between modules, threads, programs (in particular, with shared resources: sockets, files, memory, etc).
    - Runtime Exceptions
  - Off-nominal conditions: failure of some part of software of underlying machinery (network, etc).
    - Incomplete functionality
  - Deadlocks: multiple processes fighting for a resource.
    - Freeze ups, never ending processes

Debugging...

- Debugging, in general, consists of the following main stages
  - Describe the bug - Maybe this isn't part of debugging itself
  - Get the program snapshot when the bug 'appears'.
    - Try to reproduce the bug and catch the state (variables, registers, files, etc.) and action (what the program is doing at the moment, which function is running).
  - Analyze the snapshot (state/action) and search for the cause of the bug.
  - Fix the bug.
- Debugging Techniques
  - Execution tracing
    - running the program
    - Print
    - trace utilities - follows the program through the execution ; breakpoints, watchs...

Debugging...

- single stepping in debugger
- Interface checking
  - check procedure parameter number/type (if not enforced by compiler) and value
  - *defensive programming*: check inputs/results from other modules
  - documents assumptions about caller/callee relationships in modules, communication protocols, etc
- Assertions: include range constraints or other information with data.
- Skipping code: comment out suspect code, then check if error remains.

- Debugging tools (called *debugger*s) help identify coding errors at various development stages.
- Some programming language packages include a facility for checking the code for errors as it is being written.

Debugging vs testing

- Testing and debugging go together like peas in a pod:
  - Testing finds errors; debugging localizes and repairs them.
  - Together these form the "testing/debugging cycle": we test, then debug, then repeat.
  - Any debugging should be followed by a reapplication of *all* relevant tests, particularly regression tests.
    - This avoids (reduces) the introduction of new bugs when debugging.
  - Testing and debugging need not be done by the same people (and often should not be).

Software Quality Assurance

- What is “quality”?
  - Degree to which a system, component, or process meets (1) specified requirements, and (2) customer or user needs or expectations (IEEE Glossary)
  - The totality of features and characteristics of a product or service that bear on its ability to satisfy specified or implied needs (ISO)
- An alternate view of Quality
  - is not absolute
  - is multidimensional, can be difficult to quantify
  - has aspects that are not easy to measure
  - assessment is subject to constraints (e.g., cost)
  - is about acceptable compromises
  - criteria are not independent, can conflict

Software Quality Assurance...

- What is Software Quality Assurance?
  - IEEE
    - A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.
    - A set of activities designed to evaluate the process by which products are developed or manufactured.
  - "Conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software."
  - Generally,
    - Monitoring *processes* and *products* throughout the software development lifecycle to ensure the quality of the delivered product(s)

Software Quality Assurance...

- Software Quality Assurance
  - Umbrella activity applied throughout the software process
  - Planned and systematic pattern of actions required to ensure high quality in software
  - Responsibility of many stakeholders (software engineers, project managers, customers, salespeople, SQA group)

- SQA Questions
  - Does the software adequately meet its quality factors?
  - Has software development been conducted according to pre-established standards?
  - Have technical disciplines performed their SQA roles properly?

Quality Assurance Elements

- **Standards** – ensure that standards are adopted and followed
- **Reviews and audits** – audits are reviews performed by SQA personnel to ensure what quality guidelines are followed for all software engineering work
- **Testing** – ensure that testing id properly planned and conducted
- **Error/defect collection and analysis** – collects and analyses error and defect data to better understand how errors are introduced and can be eliminated
- **Changes management** – ensures that adequate change management practices have been instituted
- **Education** – takes lead in software process improvement and educational program

Quality Assurance Elements...

- **Vendor management** – suggests specific quality practices vendor should follow and incorporates quality mandates in vendor contracts
- **Security management** – ensures use of appropriate process and technology to achieve desired security level
- **Safety** – responsible for assessing impact of software failure and initiating steps to reduce risk
- **Risk management** – ensures risk management activities are properly conducted and that contingency plans have been established

SQA Tasks

- Prepare SQA plan for the project.
- Participate in the development of the project's software process description.
- Review software engineering activities to verify compliance with the defined software process.
- Audit designated software work products to verify compliance with those defined as part of the software process.
- Ensure that any deviations in software or work products are documented and handled according to a documented procedure.
- Record any evidence of noncompliance and reports them to management.