Unit I

TESTING TECHNIQUES & TEST CASE DESIGN

- Using White Box Approach to Test design
- Test Adequacy Criteria Static Testing Vs. Structural Testing
- Code Functional Testing , Coverage and Control Flow Graphs
- Covering Code Logic – Paths – Their Role in White box
- Based Test Design – Code Complexity Testing
- Evaluating Test Adequacy Criteria. Test Case Design
- Strategies – Using Black Box Approach to Test Case Design
- Random Testing – Requirements based testing
- Boundary Value Analysis –Decision tables – Equivalence Class Partitioning
- State-based testing – Cause-effect graphing – Error guessing
- Compatibility testing – User documentation testing – Domain testing
- Case study for Control Flow Graph and State-based Testing.

Testing as a Process

- The software development process has been described as a series of phases, procedures, and steps that result in the production of a software product.
- Embedded within the software development process are several other processes including testing.
- Testing itself is related to two other processes called verification and validation

- Validation is the process of evaluating a software system or component during, or at the end of the development cycle in order to determine whether it satisfies specified requirements [specification meets the customer need]
- Validation is usually associated with traditional execution-based testing, that is, exercising the code with test cases

- Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase . [software meets the specification]
- Verification is usually associated with activities such as inspections and reviews of software deliverables.

Are we building the system right?	Are we building the right system?
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	**Validation** is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.	The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.
Following activities are involved in **Verification**: Reviews, Meetings and Inspections.	Following activities are involved in **Validation**: Testing like black box testing, white box testing, gray box testing etc.
Verification is carried out by QA team to check whether implementation software is as per specification document or not.	**Validation** is carried out by testing team.
Execution of code is not comes under **Verification**.	Execution of code is comes under **Validation**.
Verification process explains whether the outputs are according to inputs or not.	**Validation** process describes whether the software is accepted by the user or not.
Verification is carried out before the Validation.	**Validation** activity is carried out just after the Verification.
Following items are evaluated during **Verification**: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc,	Following item is evaluated during **Validation**: Actual product or Software under test.
Cost of errors caught in **Verification** is less than errors found in Validation.	Cost of errors caught in **Validation** is more than errors found in Verification.
It is basically manually checking the of documents and files like requirement	It is basically checking of developed program based on the requirement specifications

- Testing itself has been defined in several ways. Two definitions are shown below
- Testing is generally described as a group of procedures carried out to evaluate some aspect of a piece of software.
- Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes

Definitions

- Many of the definitions used are based on the terms described in the IEEE Standards Collection for Software Engineering The standards collection includes the IEEE Standard Glossary of
- Software Engineering Terminology, which is a dictionary devoted to describing software engineering vocabulary
- Errors
  - An error is a mistake, misconception, or misunderstanding on the part of a software developer.
  - In the category of developer we include software engineers, programmers, analysts, and testers. For example, a developer may misunderstand a design notation, or a programmer might type a variable name incorrectly.

Faults (Defects)

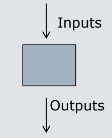
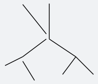
- A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.
- Faults or defects are sometimes called "bugs." Use of the latter term trivializes the impact faults have on software quality. Use of the term "defect" is also associated with software artifacts such as requirements and design documents.
- Defects occurring in these artifacts are also caused by errors and are usually detected in the review process.

- Failures
- A failure is the inability of a software system or component to perform its required functions within specified performance requirements .

Test Strategy

- The description of behavior or functionality for the software-under-test may come from a formal specification, an Input/Process/ Output Diagram (IPO), or a well-defined set of pre and post conditions.
- Another source for information is a requirements specification document that usually describes the functionality of the software-under-test and its inputs and expected outputs.
- The tester provides the specified inputs to the software-under-test, runs the test and then determines if the outputs produced are equivalent to those in the specification.

- Because the black box approach only considers software behaviour and functionality, it is often called functional, or specification-based testing.
- This approach is especially useful for revealing requirements and specification defects.
- The white box approach focuses on the inner structure of the software to be tested.
- To design test cases using this strategy the tester must have a knowledge of that structure.

Test Strategy	Tester's View	Knowledge Sources	Methods
Black box		Requirements document Specifications Domain knowledge Defect analysis Data	Equivalence class partitioning Boundary value analysis State transition testing Cause and effect graphing Error guessing
White box		High-level design Detailed design Control flow graphs Cyclomatic complexity	Statement testing Branch testing Path testing Data flow testing Mutation testing Loop testing

Using the White Box Approach to Test Case Design

- In this chapter a complementary approach to test case design will be examined where the tester has knowledge of the internal logic structure of the software under test.
- The tester's goal is to determine if all the logical and data elements in the software unit are functioning properly.
- This is called the white box, or glass box, approach to test case design.

- The knowledge needed for the white box test design approach often becomes available to the tester in the later phases of the software life cycle, specifically during the detailed design phase of development
- This is in contrast to the earlier availability of the knowledge necessary for black box test design.
- As a consequence, white box test design follows black box design as the test efforts for a given project progress in time.
- Another point of contrast between the two approaches is that the black box test design strategy can be used for both small and large software components,
- whereas white box–based test design is most useful when testing small components.

Test Adequacy Criteria

- The goal for white box testing is to ensure that the internal components of a program are working properly.
- A common focus is on structural elements such as statements and branches.
- The tester develops test cases that exercise these structural elements to determine if defects exist in the program structure.
- By exercising all of the selected structural elements the tester hopes to improve the chances for detecting defects.

- Testers need a framework for deciding which structural elements to select as the focus of testing, for choosing the appropriate test data, and for deciding when the testing efforts are adequate enough to terminate the process with confidence that the software is working properly.
- Such a framework exists in the form of test adequacy criteria
- Rules of this type can be used to determine whether or not sufficient testing has been carried out.
- The criteria can be viewed as representing minimal standards for testing a program.
- The application scope of adequacy criteria also includes:
- **(i)** helping testers to select properties of a program to focus on during test;
- **(ii)** helping testers to select a test data set for a program based on the selected properties;

```
Printme (int a, int b)
{int result = a+ b;
If (result> 0)
Print ("Positive", result)
Else Print ("Negative", result)
}
```

INPUT

A = 1, B = 1

A = -1, B = -3

The goal of WhiteBox testing in software engineering is to verify all the decision branches, loops, statements in the code.

Statement Coverage

```
Add (int a, int b) {
      If (b > a) {
   b = b - a
   Print b
          }
      If (a > b) {
      b = a – b
      Print b
}
    Else Print '0'
}
```

It is a method to ensure that each line of the source code is covered at least once by the tests.

For example, in the above source code if input values are taken as 2 & 3 then, the 'Else' part of the code would not get executed. However, if the input values are of type 3 & 2 then the 'If' part of the code would not get executed.

This means that with either set of values of our Statement Coverage would not be 100%. In such a case, we may have to execute the tests with all three [(2, 3), (3, 2), (0, 0)] set of values to ensure 100% Statement Coverage.

Branch Coverage Vs Condition Coverage

If (a >0) & (b >0)

Then Print “Hello”

Else Print “Bye”

Let us write down the data set needed for complete Branch Coverage:

(1, 1) – In this case, ‘a’ and ‘b’ both are true, so the If condition gets executed.
(1, 0) – In this case, ‘a’ is true and ‘b’ would be false, so the Else part of the code is executed.

As we know the purpose of Branch Coverage is to get every branch executed at least once and this purpose is achieved.

Condition Coverage:

(1, 0) – In this case, ‘a’ is true and ‘b’ would be false.
(0, 1) – In this case, ‘a’ is false and ‘b’ would be true.

The purpose of Condition Coverage is to get each of true and false for every condition executed and this purpose is achieved here.

Did you notice that the else part does not get executed in Condition coverage? This is where Condition Coverage differs from Branch Coverage.

- **(iii)** supporting testers with the development of quantitative objectives for testing;
- **(iv)** indicating to testers whether or not testing can be stopped for that program.
- A program is said to be adequately tested with respect to a given criterion if all of the target structural elements have been exercised according to the selected criterion.
- Using the selected adequacy criterion a tester can terminate testing when he/she has exercised the target structures, and have some confidence that the software will function in manner acceptable to the user.
- If a test data adequacy criterion focuses on the structural properties of a program it is said to be a program-based adequacy criterion.
- Program-based adequacy criteria are commonly applied in white box testing. They use either logic and control structures, data flow, program text, or faults as the focal point of an adequacy evaluation

- Other types of test data adequacy criteria focus on program specifications.
- These are called specification-based test data adequacy criteria.
- Finally, some test data adequacy criteria ignore both program structure and specification in the selection and evaluation of test data.
- Adequacy criteria are usually expressed as statements that depict the property, or feature of interest, and the conditions under which testing can be stopped (the criterion is satisfied)

- For example, an adequacy criterion that focuses on statement/branch properties is expressed as the following:

“ A test data set is statement, or branch, adequate if a test set *T* for program *P* causes all the statements, or branches, to be executed respectively. “

- In addition to statement/branch adequacy criteria as shown above, other types of program-based test data adequacy criteria are in use; for example, those based on
- (i) exercising program paths from entry to exit, and
- (ii) execution of specific path segments derived from data flow combinations such as definitions and uses of variables

- The concept of test data adequacy criteria, and the requirement that certain features or properties of the code are to be exercised by test cases, leads to an approach called “coverage analysis,” which in practice is used to set testing goals and to develop and evaluate test data.
- It follows from the requirement that the test cases developed must insure that all the statements in the unit are executed at least once.
- When a coverage related testing goal is expressed as a percent, it is often called the “degree of coverage.”
- The planned degree of coverage is usually specified as 100% if the tester wants to completely satisfy the commonly applied test adequacy, or coverage criteria.

- Under some circumstances, the planned degree of coverage may be less than 100% possibly due to the following:
- • The nature of the unit
- —Some statements/branches may not be reachable.
- —The unit may be simple, and not mission, or safety, critical, and so complete coverage is thought to be unnecessary.
- • The lack of resources
- —The time set aside for testing is not adequate to achieve 100% coverage.
- —There are not enough trained testers to achieve complete coverage for all of the units.
- —There is a lack of tools to support complete coverage.
- Other project-related issues such as timing, scheduling, and marketing constraints

- The following scenario is used to illustrate the application of coverage analysis.
  - Suppose that a tester specifies “branches” as a target property for a series of tests.
- This could be specified in the test plan as a requirement for 100% branch coverage for a software unit under test.
- In this case the tester must develop a set of test data that insures that all of the branches (true/false conditions) in the unit will be executed at least once by the test cases.
- When the planned test cases are executed under the control of a coverage tool, the actual degree of coverage is measured.
- If there are, for example, four branches in the software unit, and only two are executed by the planned set of test cases, then the degree of branch coverage is 50%.

Coverage and Control Flow Graphs

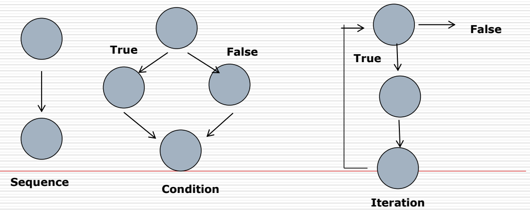
- The application of coverage analysis is typically associated with the use of control and data flow models to represent program structural elements and data. The logic elements most commonly considered for coverage are based on the flow of control in a unit of code. For example,
- **(i)** program statements;
- **(ii)** decisions/branches (these influence the program flow of control);
- **(iii)** conditions (expressions that evaluate to true/false, and do not contain any other true/false-valued expressions);

  (iv) combinations of decisions and conditions;
- **(v)** paths (node sequences in flow graphs).

These logical elements are rooted in the concept of a program prime. A program prime is an atomic programming unit. All structured programs can be built from three basic e.g., primes-

- sequential (assignment statements)
- decision (e.g., if/then/else statements),
- iterative (e.g., while, for loops).

Graphical representations for these three primes are shown in Figure



- □ Using the concept of a prime and the ability to use combinations of primes to develop structured code, a (control) flow diagram for the software unit under test can be developed.
- □ The flow graph can be used by the tester to evaluate the code with respect to its testability, as well as to develop white box test cases
- □ There are commercial tools that will generate control flow graphs from code and in some cases from pseudo code.

- The tester can use tool support for developing control flow graphs especially for complex pieces of code.
- A control flow representation for the software under test facilitates the design of white box–based test cases as it clearly shows the logic elements needed to design the test cases using the coverage criterion of choice.

Static Testing Vs Structural Testing

- Static testing
  - Static testing is a software testing method that involves examination of program's code and its associated documentation but does not require the program to be executed.
  - Also called as Dry-Run Testing.
  - Requires programmers to manually read their own code to find any errors. Hence named 'static'.
  - Static testing is a stage of White Box Testing.

- Main objective of this testing is to improve the quality of software products by finding errors in early stages of the development cycle.
- Most static testing techniques can be used to 'test' any form of document including source code, design documents and models, functional specifications and requirement specifications
- Participants in Static Testing During a review four types of participants take part.
- They are:
  1. Moderator
  2. Author
  3. Scribe
  4. Reviewer
  5. Manager

- The moderator Also known as review leader
- Performs entry check
- Follow-up on the rework
- Schedules the meeting
- Coaches other team
- Leads the possible discussion and stores the data that is collected.
- The author Illuminate the unclear areas and understand the defects found.
- Basic goal should be to learn as much as possible with regard to improving the quality of the document.
- The scribe is a separate person to do the logging of the defects found during the review.

- The reviewers Also known as checkers or inspectors
- Check any material for defects, mostly prior to the meeting
- The manager can also be involved in the review depending on his or her background.
- Manager decides on the execution of reviews.
- Allocates time in project schedules and determines whether review process objectives have been met

Static Testing Techniques

- Informal Reviews
- Formal Reviews
- Technical Reviews
- Walk Through
- Inspection Process
- Static Code Review.

- Informal Review
- Doesn't follow any process to find errors in the document, you just review the document and give informal comments on it.
- Applied many times during the early stages of the life cycle of the document.
- A two person team can conduct an informal review and in later stages more people are involved.
- The goal is to keep the author and to improve the quality of the document.
- The most important thing to keep in mind about the informal reviews is that they are not documented.

Formal Review

Formal reviews follow a formal process.

It is well structured and regulated (Controlled).

A formal review process consists of six main steps

1. Planning
2. Kick-off
3. Preparation
4. Review meeting
5. Rework
6. Follow-up

Technical Review

- A team consisting of your peers, review the technical specification of the software product and checks whether it is suitable for the project.

- They try to find any discrepancies in the specifications and standards followed.
- This review concentrates mainly on the technical document related to the software such as Test Strategy,
- Walkthrough
- Test Plan and requirement specification documents.

The author of the work product explains the product to his team.

- Participants can ask questions if any.
- Meeting is led by the author.
- Scribe makes note of review comments

- **Inspection**
- The main purpose is to find defects and meeting is led by trained moderator.
- This review is a formal type of review where it follows strict process to find the defects.
- Reviewers have checklist to review the work products .
- They record the defect and inform the participants to rectify those errors.
- **Static Code Review**
- This is systematic review of the software source code without executing the code.
- It checks the syntax of the code, coding standards, code optimization, etc.
- This is also termed as white box testing.

□ Advantages of Static Testing

- Since static testing can start early in the life cycle so early feedback on quality issues can be established.
- As the defects are getting detected at an early stage so the rework (Revise and rewrite) cost most often relatively low.
- Development productivity is likely to increase because of the less rework effort.

□ Disadvantages of Static Testing

- Time consuming as conducted manually.
- Does not find vulnerabilities introduced in runtime environment.
- Limited trainee personnel to thoroughly conduct static code analysis.

STRUCTURAL (WHITE-BOX) TESTING

Outline

□ Path Testing

□ Control Flow Graph

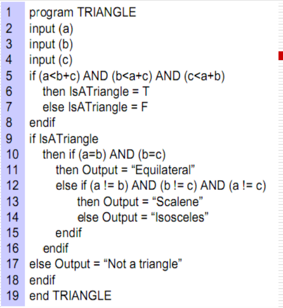
□ DD-Path Graph

Path testing

□ Structural testing method

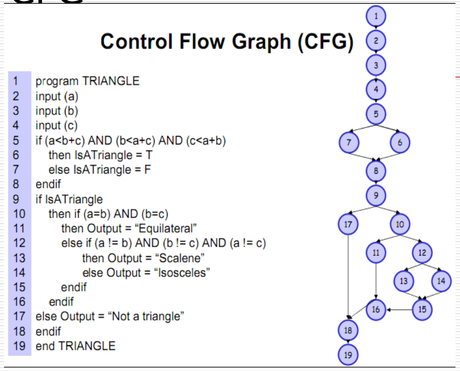
□ Based on the source code / pseudocode of the program or the system, and NOT on its specification

Example: Find the Control Flow Graph (CFG) for the Triangle Problem



```
program TRIANGLE
input (a)
input (b)
input (c)
if (a<b+c) AND (b<a+c) AND (c<a+b)
  then IsATriangle = T
  else IsATriangle = F
endif
if IsATriangle
  then if (a=b) AND (b=c)
    then Output = “Equilateral”
    else if (a != b) AND (b != c) AND (a != c)
      then Output = “Scalene”
      else Output = “Isosceles”
    endif
  endif
else Output = “Not a triangle”
endif
end TRIANGLE
```

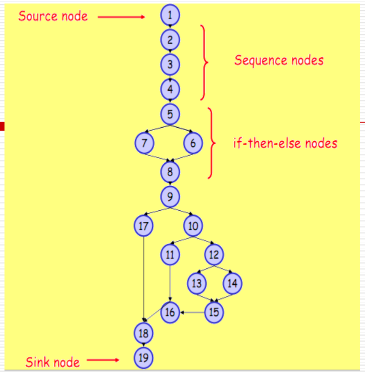
CFG



Control Flow Graph (CFG)

```
program TRIANGLE
input (a)
input (b)
input (c)
if (a<b+c) AND (b<a+c) AND (c<a+b)
  then IsATriangle = T
  else IsATriangle = F
endif
if IsATriangle
  then if (a=b) AND (b=c)
    then Output = “Equilateral”
    else if (a != b) AND (b != c) AND (a != c)
      then Output = “Scalene”
      else Output = “Isosceles”
    endif
  endif
else Output = “Not a triangle”
endif
end TRIANGLE
```

CFG



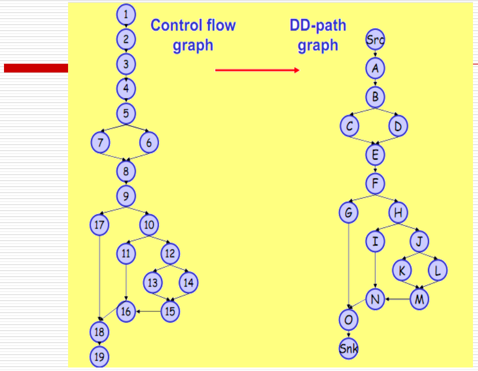
DD – Path Graph

- □ A CFG can be broken into DD-paths
- □ The resulting graph is called a DD-path graph of the program

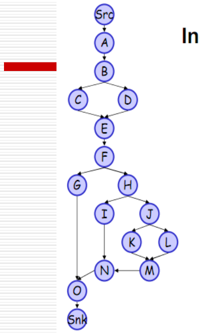
DD – Path Graph

Node	DD-Path Name
1	Source
2 - 4	A
5	B
6	C
7	D
8	E
…	…
19	Sink

CFG to DD-Path



Independent (basis) paths



Independent paths

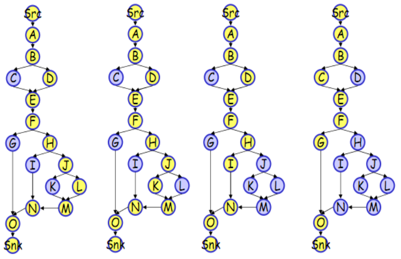
Find at least one independent path in the graph

P1: Src-A-B-D-E-F-H-J-K-M-N-O-Snk
P2: Src-A-B-D-E-F-H-J-L-M-N-O-Snk
P3: Src-A-B-D-E-F-H-I-N-O-Snk
P4: Src-A-B-C-E-F-G-O-Snk

P1: Scalene
P2: Isosceles
P3: Equilateral
P4: Not a triangle

Independent (basis) paths

Src-A-B-D-E-F-H-J-K-M-N-O-Snk
Src-A-B-D-E-F-H-J-L-M-N-O-Snk
Src-A-B-D-E-F-H-I-N-O-Snk
Src-A-B-C-E-F-G-O-Snk



Code Functional Testing

- •Motivation
- •Example
- •Basic Methods

Timing: 30 minutes

Code Functional Testing: Motivation

- □With structural testing, we can guarantee all code has been executed with tests
- □But say we have missed, ignored or forgotten some requirements
  - ■ Those requirements may not have been implemented in code
  - ■ No code → nothing to cover!
- □Therefore, we can get 100% coverage *without* actually satisfying the requirements
- □We need to test based on the requirements (functionality) too
  - ■ This is *functional* testing
  - ■ We will need to also consider "unstated" requirements like error handling
- □We would like to go *systematically* through all stated and unstated requirements
  - ■ There are standard ways of doing this

Functional Testing Example

- □Simple "triangle analyzer"
- □Requirements:
  - ■ Program prompts user for input
  - ■ User types in three real numbers separated by commas
    - □ E.g. 2.5, 6, 6.5
  - ■ Numbers are supposed to be lengths of the sides of a triangle
  - ■ Program responds with:
    - □ ***Equilateral*** – if there is a valid triangle with those side lengths and that type
    - □ ***Isosceles*** – likewise
    - □ ***Scalene*** – likewise
    - □ ***Not a triangle*** – if there is no valid triangle with those side lengths (e.g. 3, 4, 1000)
- □What test cases to use?
  - ■ These will be ***functional*** (black-box) test cases, since we are working only from the requirements

Triangle Program: Our Test Cases

```
1   program TRIANGLE
2   input (a)
3   input (b)
4   input (c)
5   if (a<b+c) AND (b<a+c) AND (c<a+b)
6       then IsATriangle = T
7       else IsATriangle = F
8   endif
9   if IsATriangle
10      then if (a=b) AND (b=c)
11          then Output = "Equilateral"
12          else if (a != b) AND (b != c) AND (a != c)
13                  then Output = "Scalene"
14                  else Output = "Isosceles"
15          endif
16      endif
17  else Output = "Not a triangle"
18  endif
19  end TRIANGLE
```

Functional Testing: Basic Methods

- How to choose good functional test cases?
- We will look at each of the following methods:
  - Test all possible outputs
  - Test both valid and invalid inputs
  - Test around boundaries
  - Test extreme values
  - Test input syntax
  - Guess at possible errors
- These techniques will generate lots and lots of test cases
- In general:
  - A test case may be one input or a sequence of inputs (depending on program)
  - We will probably have more test cases for *erroneous* input than correct input

Test All Possible Outputs

- For each possible kind of output that we know the program could produce:
  - Write a test case that will produce that kind of output
  - Called equivalence classes in our textbook

Examples:

- Triangle program
  - One case for each of Equilateral, Isosceles, Scalene and Not a triangle
- Parking garage simulator:
  - Should have a case (or cases) for parking a car and retrieve a car
  - Should also have cases for "garage full", "no such car"
- What do we mean by "the same kind" of output
  - This is a judgment call
  - We have to just make some reasonable choice

- We can start by choosing one test case producing each kind of output

Test Valid and Invalid Inputs

- Often, individual inputs x to a program have:
  - Valid *ranges*; e.g. x >= 0, x is from 1 to 12
  - Valid *sets of values;* e.g.
    - X is a string of alphanumeric characters
    - X ∈ {red, green, blue}
- Inputs outside these ranges or sets are invalid
- For each input to the program:
  - Test at least one "valid" value
  - Test at least one "invalid" value
  - Test invalid values off either end of value range, if appropriate

Individual inputs to the program" can include:

- Things typed in to a text interface or GUI
- Command-line options

Examples of invalid inputs:

- Triangle program:
  - → Test –1 or Z as length of side
- Day planner program:
  - → Enter Jqx as name of a month

Test Around Boundaries

- Failure often occur close to "boundaries"; e.g.
  - Boundaries between different kinds of output
  - Boundaries between valid and invalid inputs
- These failures are due to faults like
  - Small errors in arithmetic
  - Using <= instead of <
  - Not initializing a loop properly
- Therefore, should test at or near boundaries

Examples:

Triangle program:

- Test case : 2, 2, 3.999 (almost but not a triangle)
- Test case : 2, 2, 4 (right on boundary)

Pop machine program:

- What happens if user has just exactly enough money in account to buy one can of pop
- Should be allowed
- However, if the test in the program is something like

if (balance > cost), it will not be allowed

It should be: **if (balance >= cost)**

Therefore, need to test this situation

Test Extreme Values

- Often, software does not handle "very large" or "very small" values correctly
  - These failures are due to things like:
    - Buffer or arithmetic overflows
    - Mistaken assumptions that a string will be non-empty
  - Therefore those values may be a way to break the program
- Examples
  - With just about any program that takes user input:
    - Empty strings
    - Very long strings
  - Triangle Program:
    - Test case: 4321432134, 543234344, 6566765888 (very large)
    - Test case: 0.00000003, 0.00000008, 0.000000005 (very small)

Test Input Syntax

- What happens if input syntax wrong? e.g.
  - Something left out
    - `5, 12 13` → comma missing
  - Too few / too many repetitions
    - `5, 12` or `5, 12, 13, 20`
  - Invalid tokens
    - `5, 12, axolot!`
- In these situations
  - Program shouldn't just crash
  - Shouldn't give an uninformative error message
  - Also, program shouldn't just accept and process as if correct!
    - This could even be worst
  - Should give an informative error message, recover from error

Guess at Faults

- Finally, use intuition to think of how program *might* be wrong
  - Might be better to get person A to think of possible faults in person B's code
- Example with the triangle program:
  - To see whether triangle isosceles, the code must try all three distinct pairs amongst the three numbers for equality
  - What if not all three pairs have been tried by this code?
  - Therefore test all three of "2,2,3", "2,3,2", "3,2,2"
- This method not based on systematic study
  - More like educated guesses

Building a Test Suite

- □Tests from all the previous categories can be helpful
  - ■ Sometimes a given test case might be considered to be in several of these categories
- □Once we think of a good test case, write it down!
- □Recorded test suites can be re-run as software changes.

Exercise

- □ Assume you are writing a driver to test the following function:
- □ `public int maxOfThreeNumbers(int n1, int n2, int n3)`

ghly

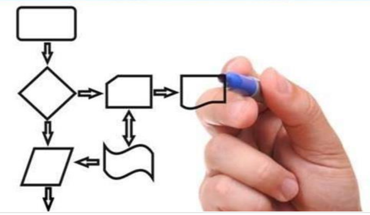
n1	n2	n3	Expected Output
99999999	999999999	999999999	Error, throws error perhaps?

Overview - Complexity..

- ✓ What is Software Design?
- ✓ Software Metrics.
- ✓ Complexity.
- ✓ Cyclomatic Complexity.
- ✓ Example.
- ✓ Determine Cyclometic Complexity.
- ✓ How this metrics is useful for software Testing?
- ✓ Uses of Cyclometic Complexity.
- ✓ Conclusion.

What is a Software Design?

❑Software Design is a process to transform user requirements into some suitable form which helps the programmer in software coding and implementation.



Is Software Metrics Required...

❑To improve quality of a software, we need to measure various aspects:-

- ➢ Size,
- ➢ Complexity,
- ➢ Maintainability of software etc..

❑Choosing the right metrics can make the difference between success and failure of a project.

Complexity...

❑The Term Complexity Stands for state of events or things, which have multiple interconnected links and highly complicated structures.

Cont..

❑ Complexity can be interpreted in different ways:-

- ➢ Problem complexity (also called computational complexity) measures the complexity of the underlying problem.
- ➢ Algorithmic complexity reflects the complexity of the algorithm implemented to solve the problem.
- ➢ Structural complexity measures the structure of the software used to implement the algorithm.
- ➢ Cognitive complexity measures the effort required to understand the software.

Cyclometic Complexity..

- ❑ A program consists of statements.
- ❑ Some of them are decision making which change the flow of program.
- ❑ Developed by McCabe, in 1976.
- ❑ Measures the number of linearly independent paths through a program.
- ❑ Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand.

Cont..

❑The complexity would be 1, since there would be only a single path through the code.

❑ If the code had one single-condition IF statement, there would be two paths through the code, so the complexity would be 2.

Determine Cyclometic Complexity..

There are several methods:

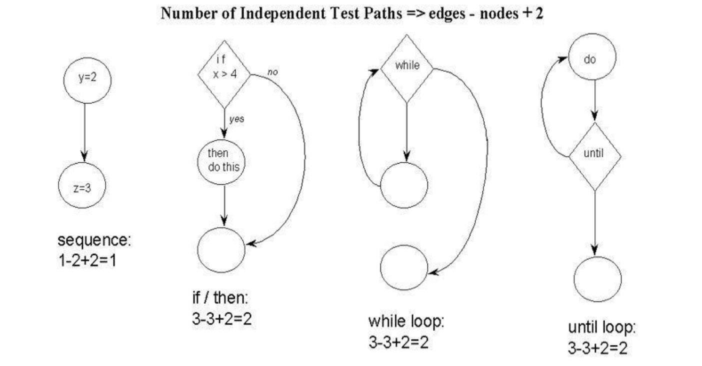
1.Cyclomatic complexity = edges - nodes + 2

2.Cyclomatic complexity = Number of Predicate Nodes + 1

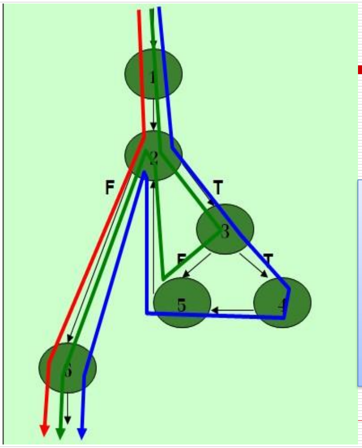
3.Cyclomatic complexity = Number of regions in the control flow graph

Example..

Cyclometic Complexity = Edges –Nodes + 2

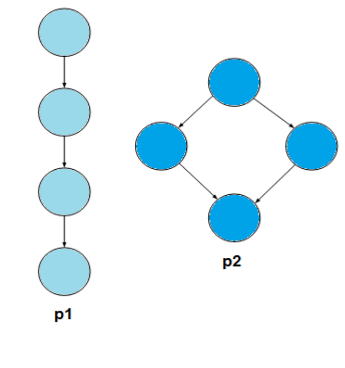


Another Example..



- □ In this Example:-
  - ❑ [1—2—6]
  - ❑ [1—2—3—5—2—6]
  - ❑ [1—2—3—4—5—2—6]
- □ Cyclomatic Complexity
- □ = 7 - 6 + 2*1 = 3

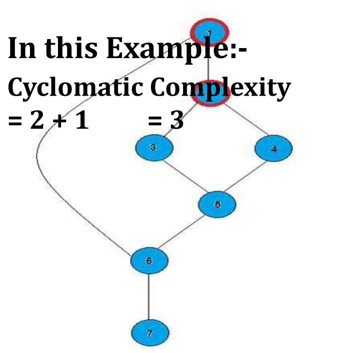
Cont..



- □ In this Example:-
- □ Cyclomatic Complexity
- □ = 7 - 8 + 2*2 = 3

Cont..

Cyclomatic complexity= Number of Predicate Nodes + 1

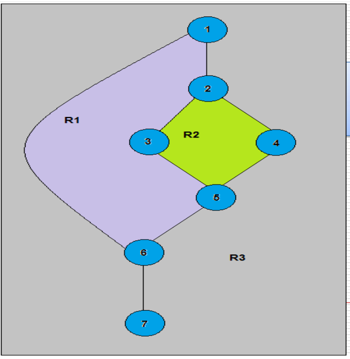


In this Example:-

Cyclomatic Complexity = 2 + 1 = 3

Cont..

Cyclomatic complexity = number of regions in the control flow graph.



How this metric is useful for software testing..

❑Basis Path testing is one of White box technique and it guarantees to execute at least one statement during testing.

❑It checks each linearly independent path through the program.

Cont..

❑Which means number of test cases, will be equivalent to the cyclomatic complexity of the program.

Cyclomatic Complexity	Risk Evaluation	Probability of Bad fix
1-10	Low risk, testable code	5%
11-20	Moderate Risk	10%
21-50	High Risk	30%
>50	Very High Risk, untestable code	40%

Cont..

❑The advantage of this is a function can have maximum test case of 10.

❑And research has proven that most computer programmers can easily read and modify functions that are having Cyclomatic complexity less than or equal to 10.

❑As cognitive load on human mind is less.

Uses of Cyclomatic Complexity..

Cyclomatic Complexity can prove to be very helpful in :-

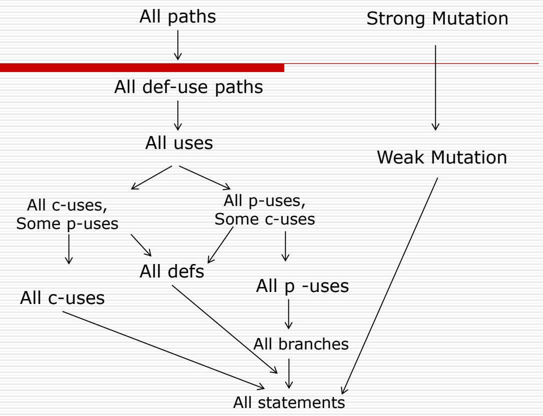
- Helps developers and testers to determine independent path executions.
- Developers can assure that all the paths have been tested at least once.
- Helps us to focus more on the uncovered paths.
- Evaluate the risk associated with the application or program.

Conclusion..

- It is mainly used to evaluate complexity of a program.
- If the decision points are more, then complexity of the program is more.
- If program has high complexity number, then probability of error is high with increased time for maintenance and trouble shoot.

Evaluating Test Adequacy Criteria

- Most of the white box testing approaches are associated with application of an adequacy criterion
- Tester often faced with the decision of which criterion to apply to a given item under test given the nature of the item and constraints of the test environment (time, costs, resources)
- One source of information the tester can use to select an appropriate criterion is the test adequacy criterion hierarchy as shown in the figure which gives a greater thoroughness in testing.



A partial ordering for test adequacy criteria

- The criteria at the top of the hierarchy are said to subsume those at the lower levels.
- In Many organizations achieving a high level of statement coverage is not even as a minimal testing goal.
- As a tester we might at first reason the goal to develop tests that can satisfy the most stringent criterion.
- Application so called "stronger" criteria usually requires more tester time and resources.

- Support for evaluating test adequacy criteria comes from a theory developed by Weyuker. She presents a set of axioms that allow testers to formalize properties to be satisfied by any program-based adequacy criterion.
- Testers can use the axioms to
- recognize both strong and weak adequacy criteria; a tester may decide to use a weak criterion, but should be aware of its weakness with respect to the properties described by the axioms;

- focus attention on the properties that an effective test data adequacy criterion should exhibit;
- select an appropriate criterion for the item under test;
- stimulate thought for the development of new criteria; the axioms are the framework with which to evaluate these new criteria.

- The axioms are based on the following set of assumptions [2]:
- **(i)** programs are written in a structured programming language;
- **(ii)** programs are SESE (single entry/single exit);
- **(iii)** all input statements appear at the beginning of the program;
- **(iv)** all output statements appear at the end of the program.

- Applicability Property
  - For every program there exists an adequate test set." What this axiom means is that for all programs we should be able to design an adequate test set that properly tests it.
  - The test set may be very large so the tester will want to select representable points [applicable property] of the specification domain to test it.
  - If we test on all representable points, that is called an exhaustive test set.
  - The exhaustive test set will surely be adequate since there will be no other test data that we can generate.
  - However, in past discussions we have ruled out exhaustive testing because in most cases it is too expensive, time consuming, and impractical.

- □ Nonexhaustive Applicability Property
  - ■ "For a program P and a test set T, P is adequately tested by the test set T, and T is not an exhaustive test set."
  - ■ To paraphrase, a tester does not need an exhaustive test set in order to adequately test a program.
- □ Monotonicity Property
  - ■ "If a test set T is adequate for program P, and if T is equal to, or a subset of T1 , then T is adequate for program P."
- □ Inadequate Empty Set
  - ■ "An empty test set is not an adequate test for any program." If a program is not tested at all, a tester cannot claim it has been adequately tested!

- □ Antiextensionality Property
  - ■ There are programs P and Q such that P is equivalent to Q, and T is adequate for P, but T is not adequate for Q."
  - ■ We can interpret this axiom as saying that just because two programs are semantically equivalent (they may perform the same function) does not mean we should test them the same way.
  - ■ Their implementations (code structure) may be very different.
- □ General Multiple Change Property
  - ■ There are programs *P* and *Q* that have the same shape, and there is a test set *T* such that *T* is adequate for *P,* but is not adequate for *Q.*"
  - ■ Here Weyuker introduces the concept of shape to express a

- □ She states that two programs are the same shape if one can be transformed into the other by applying the set of rules shown below any number of times:
  - ■ (i) replace relational operator r1 in a predicate with relational operator r2;
  - ■ (ii) replace constant c1 in a predicate of an assignment statement with constant c2;
  - ■ (iii) replace arithmetic operator a1 in an assignment statement with arithmetic operator a2.
- □ Axiom 5 says that semantic closeness is not sufficient to imply that two programs should be tested in the same way.
- □ Given this definition of shape, Axiom 6 says that even the syntactic closeness of two programs is not strong enough reason to imply they should be tested in the same way.

- □ Antidecomposition Property
  - ■ "There is a program P and a component Q such that T is adequate for P, T is the set of vectors of values that variables can assume on entrance to Q for some t in T, and T is not adequate for Q."
  - ■ This axiom states that although an encompassing program has been adequately tested, it does not follow that each of its components parts has been properly tested.
  - ■ Implications for this axiom are:
    - □ a routine that has been adequately tested in one environment may not have been adequately tested to work in another environment, the environment being the enclosing program.

- □ Anticomposition Property
  - ■ "There are programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P; Q (the composition of P and Q)."
  - ■ Paraphrasing this axiom we can say that adequately testing each individual program component in isolation does not necessarily mean that we have adequately tested the entire program

- □ Renaming Property
  - ■ If P is a renaming of Q, then T is adequate for P only if T is adequate for Q. A program P is a renaming of Q if P is identical to Q except for the fact that all instances of an identifier, let us say a in Q have been replaced in P by an identifier, let us say b, where "b" does not occur in Q, or if there is a set of such renamed identifiers."
  - ■ This axiom simply says that an inessential change in a program such as changing the names of the variables should not change the nature of the test data that are needed to adequately test the program.

- □ Complexity Property
  - ■ "For every n, there is a program P such that P is adequately tested by a size n test set, but not by any size n - 1 test set."
  - ■ This means that for every program, there are other programs that require more testing.
- □ Statement Coverage Property
  - ■ If the test set *T* is adequate for *P,* then *T* causes every executable statement of *P* to be executed."
  - ■ Ensuring that their test set executed all statements in a program is a minimum coverage goal for a tester.

- □ Random Testing
  - ■ Each Software Module or system has an input domain from which input data is selected.
  - ■ If a tester randomly selects inputs from the domain, this is called random testing.
  - ■ For ex: if the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsymmetrically, select values from within that domain; Ex : 55,24,3 might be chosen
  - ■ But there are some issues
    - □ Are the three values adequate to show that the module meets the specification? Should additional or fewer values be used to make the most effective use of resources ?

- ■ Are there any input values, other than those selected, more like to reveal defects ? (Like positive integers at the beginning or end of the domain)
- ■ Should any values outside the valid domain be used as test inputs ?
- □ Use of random test inputs may save some time and effort
- □ However, the tester should keep in mind that according to many testing experts, selecting test inputs randomly has very little chance of producing an effective set of test data.
- □ As a note that there are tools that generate random test data for stress tests. Useful at system level

Equivalence Class Partitioning

□ If a tester is viewing the software as a black box with well defined inputs and outputs, a good approach to selecting test inputs is to use a method called equivalence class partitioning.

□ Equivalence class partitioning results in a partitioning of the input domain of the software- under-test.

□ This technique can also be used to partition the output domain, but not a common usage.

□ The finite no. of partitions or equivalence classes result allow the tester to select a given member of an equivalence class as a representative of that class.

□ Using equivalence class partitioning a test value in a particular class is equivalent to a test value of any other member of that class.

□ If one test case in a particular equivalence class reveals a defect, the other test cases based on that class would be expected to reveal the same.

□ Partitioning of the input domain for the software-under-test using this technique has the following advantages :

- ■ It eliminates the need for exhaustive testing, which is not feasible.
- ■ It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect
- ■ It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class.

□ How does the tester identify equivalence classes for the input domain ?

□ One approach is to use a set of what Glen Myers calls "interesting" input conditions.

□ The tester and analyst interact during the analysis phase to develop

- ■ (i) a set of testable requirements,
- ■ (ii) correct and complete input/output specification

□ From these the tester develops,

- ■ (i) a high-level test plan,
- ■ A preliminary set of black box test cases for the system.

□ There are several important points related to equivalence class partitioning

- ■ The test must consider both valid and invalid equivalence classes
- ■ Equivalence classes may also be selected for output conditions.
- ■ The derivations of input or output equivalence classes is a heuristic process.

□ In some cases, it is difficult for the tester to identify equivalence classes. The conditions/boundaries that help to define classes may be absent, or there may seem to be a very large or very small number of equivalence classes for the problem domain.

□ Myers suggests the following conditions as guidelines for selecting input equivalence classes

□ List of Conditions

- □ "If an input condition for the software-under-test is specified as a *range of values*, select one valid equivalence class that covers the allowed range and two invalid equivalence classes, one outside each end of the range."
- ■ Ex : input, the length of a widget in millimetres, lies in the range 1–499; then select one valid equivalence class that includes all values from 1 to 499. Select a second equivalence class that consists of all values less than 1, and a third equivalence class that consists of all values greater than 499.

□ "If an input condition for the software-under-test is specified as a *number of values*, then select one valid equivalence class that includes the allowed number of values and two invalid equivalence classes that are outside each end of the allowed number."

□ Ex : if the specification for a real estate-related module say that a house can have one to four owners, then we select one valid equivalence class that includes all the valid number of owners, and then two invalid equivalence classes for less than one owner and more than four owners

□ "If an input condition for the software-under-test is specified as a *set* of valid input values, then select one valid equivalence class that contains all the members of the set and one invalid equivalence class for any value outside the set."

- ■ For example, if the specification for a paint module states that the colors RED, BLUE, GREEN and YELLOW are allowed as inputs, then select one valid equivalence class that includes the set RED, BLUE, GREEN and YELLOW, and one invalid equivalence class for all other inputs.

□ "If an input condition for the software-under-test is specified as a "*must be*" condition, select one valid equivalence class to represent the "must be" condition and one invalid class that does not include the "must be" condition."

□ "If the input specification or any other information leads to the belief that an element in an equivalence class is not handled in an identical way by the software-under-test, then the class should be further partitioned into smaller equivalence classes."

- Example :
  - This is a specification for a module that calculates a square root.

Function square_root

message (x:real)

when x >=0.0

reply (y:real)

where y >=0.0 & approximately (y*y,x)

otherwise reply exception imaginary_square_root

end function

- The specification describes for the tester conditions relevant to the input/output variables x and y.

- The input conditions are that the variable x must be a real number and be equal to or greater than 0.0.
- The conditions for the output variable y are that it must be a real number equal to or greater than 0.0, whose square is approximately equal to x.
- If x is not equal to or greater than 0.0, then an exception is raised.
- From this information the tester can easily generate both invalid and valid equivalence classes and boundaries.

- For example, input equivalence classes for this module are the following:
- EC1. The input variable *x* is real, valid.
- EC2. The input variable *x* is not real, invalid.
- EC3. The value of *x* is greater than 0.0, valid.
- EC4. The value of *x* is less than 0.0, invalid

- Boundary Value Analysis
  - The test cases developed based on equivalence class partitioning can be strengthened by use of an another technique called boundary value analysis
  - With experience, testers soon realize that many defects occur directly on, and above and below, the edges of equivalence classes.
  - Test cases that consider these boundaries on both the input and output spaces are often valuable in revealing defects.

- boundary value analysis requires that the tester select elements close to the edges, so that both the upper and lower edges of an equivalence class are covered by test cases.
- The rules-of-thumb described below are useful for getting started with boundary value analysis.
  - If an input condition for the software-under-test is specified as a *range* of values, develop valid test cases for the ends of the range, and invalid test cases for possibilities just above and below the ends of the range.
  - If an input condition for the software-under-test is specified as a *number* of values, develop valid test cases for the minimum and maximum numbers as well as invalid test cases that include one lesser and one greater than the maximum and minimum

- If the input or output of the software-under-test is an ordered set, such as a table or a linear list, develop tests that focus on the first and last elements of the set.

- An Example of the Application of Equivalence Class Partitioning and Boundary Value Analysis
  - user to enter new widget identifiers into a widget data base. We will focus only on selecting equivalence classes and boundary values for the inputs.
  - The input specification for the module states that a widget identifier should consist of 3–15 alphanumeric characters of which the first two must be letters

- We have three separate conditions that apply to the input:
- (i) it must consist of alphanumeric characters, (ii) the range for the total number of characters is between 3 and 15, and
- (iii) the first two characters must be letters.
- We will label the equivalence classes with an identifier ECxxx, where xxx is an integer whose value is one or greater. Each class will also be categorized as valid or invalid for the input domain.

- First we consider condition 1, the requirement for alphanumeric characters.
- This is a "must be" condition.
- We derive two equivalence classes.
  - EC1. Part name is alphanumeric, valid.
  - EC2. Part name is not alphanumeric, invalid.
- Then we treat condition 2, the range of allowed characters 3–15.
  - EC3. The widget identifier has between 3 and 15 characters, valid.
  - EC4. The widget identifier has less than 3 characters, invalid.
  - EC5. The widget identifier has greater than 15 characters, invalid.
- Finally we treat the "must be" case for the first two characters.
  - EC6. The first 2 characters are letters, valid.
  - EC7. The first 2 characters are not letters, invalid.

- The equivalence classes selected may be recorded in the form of a table as shown in Table

Condition	Valid equivalence Classes	Invalid equivalence classes
1	EC1	EC2
2	EC3	EC4,EC5
3	EC6	EC 7

- Boundary value analysis is now used to refine the results of equivalence class partitioning.
- The boundaries to focus on are those in the allowed length for the widget identifier
- For example:
- **BLB—**a value just below the lower bound
- **LB—**the value on the lower boundary
- **ALB—**a value just above the lower boundary
- **BUB—**a value just below the upper bound
- **UB—**the value on the upper bound
- **AUB—**a value just above the upper bound
- For our example module the values for the bounds groups are:
- **BLB—**2 **BUB—**14
- **LB—**3 **UB—**15
- **ALB—**4 **AUB—**16

Test case Identifier	Input Values	Valid equivalence classes and bounds Covered	Invalid equivalence classes and bounds covered
1	abc1	EC1, EC3(ALB) EC6	
2	ab1	EC1, EC3(LB), EC6	
3	abcdef123456789	EC1, EC3 (UB) EC6	
4	abcde123456789	EC1, EC3 (BUB) EC6	
5	abc*	EC3(ALB), EC6	EC2
6	ab	EC1, EC6	EC4(BLB)
7	abcdefg123456789	EC1, EC6	EC5(AUB)
8	a123	EC1, EC3 (ALB)	EC7
9	abcdef123	EC1, EC3, EC6	

Summary of test inputs using equivalence class partitioning and boundary value analysis for sample module.

- Cause-and-Effect Graphing
- A major weakness with equivalence class partitioning is that it does not allow testers to combine conditions
- Combinations can be covered in some cases by test cases generated from the classes.
- Cause-and-effect graphing is a technique that can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification.
- The specification must be transformed into a graph that resembles a digital logic circuit

- The tester is not required to have a background in electronics, but he should have knowledge of Boolean logic.
- The graph itself must be expressed in a graphical language
- Developing the graph, especially for a complex module with many combinations of inputs, is difficult and time consuming.
- The graph must be converted to a decision table that the tester uses to develop test cases.
- Tools are available for the latter process and allow the derivation of test cases to be more practical using this approach

- The steps in developing test cases with a cause-and-effect graph are as follows
  - The tester must decompose the specification of a complex software component into lower-level units.
  - For each specification unit, the tester needs to identify causes and their effects. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation. Putting together a table of causes and effects helps the tester to record the necessary details. The logical relationships between the causes and effects should be determined. It is useful to express these in the form of a set of rules

- From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects. Causes are placed on the left side of the graph and effects on the right. Logical relationships are expressed using standard logical operators such as AND, OR, and NOT, and are associated with arcs. An example of the notation is shown in of graph notations
- The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
- The graph is then converted to a decision table.
- The columns in the decision table are transformed into test cases.

- The following example illustrates the application of this technique.
- Suppose we have a specification for a module that allows a user to perform a search for a character in an existing string.
- The specification states that the user must input the length of the string and the character to search for.
- If the string length is out-of-range an error message will appear
- If the character appears in the string, its position will be reported
- If the character is not in the string the message "not found" will be output.

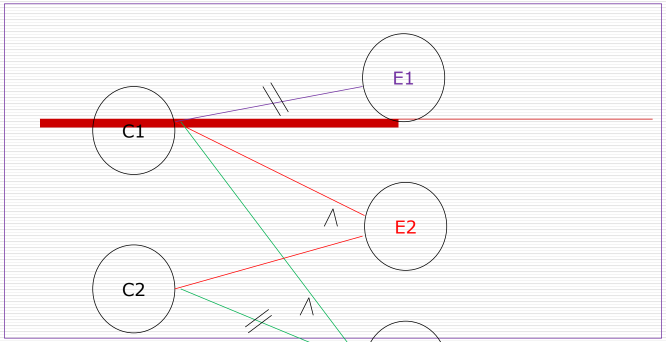
- The input conditions, or causes are as follows:
  - C1: Positive integer from 1 to 80
  - C2: Character to search for is in string
  - The output conditions, or effects are:
  - E1: Integer out of range
  - E2: Position of character in string
  - E3: Character not found
- The rules or relationships can be described as follows:
  - If C1 and C2, then E2.
  - If C1 and not C2, then E3.
  - If not C1, then E1.
- The next step is to develop a decision table. The decision table reflects the rules and the graph and shows the effects for all possible combinations of causes.

Inputs	Length	Character to search for	Outputs
T1	5	C	3
T2	5	W	Not found
T3	90		Integer out of range

- Columns list each combination of causes, and each column represents a test case.
- In this example, since we have only two causes, the size and complexity of the decision table is not a big problem.
- However, with specifications having large numbers of causes and effects the size of the decision table can be large.
- A decision table will have a row for each cause and each effect.
- The entries are a reflection of the rules and the entities in the cause and effect graph
- Entries in the table can be represented by a "1" for a cause or effect that is present, a "0" represents the absence of a cause or effect, and a "—" indicates a "don't care" value

- A decision table for our simple example is shown in Table 3 where C1, C2, C3 represent the causes, E1, E2, E3 the effects, and columns T1, T2, T3 the test cases.

	T1	T2	T3
C1	1	1	0
C2	1	0	-
E1	0	0	1
E2	1	0	0
E3	0	1	0

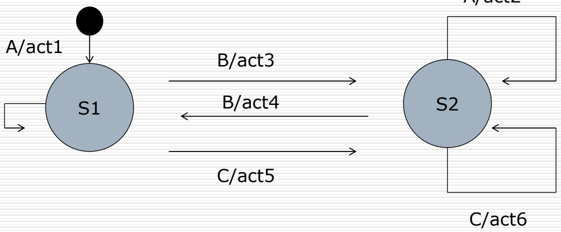


If C1 and C2, then E2.
If C1 and not C2, then E3.
If not C1, then E1

- **State Transition Testing**
  - State transition testing is useful for both procedural and object-oriented development.
  - It is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes.
  - This view gives the tester an additional opportunity to develop test cases to detect defects that may not be revealed using the input/output condition as well as cause-and-effect views presented by equivalence class partitioning and cause-and-effect graphing

- A state is an internal configuration of a system or component. It is defined in terms of the values assumed at a particular time for the variables that characterize the system or component.
- A finite-state machine is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states.
- During the specification phase a state transition graph (STG) may be generated for the system as a whole and/or specific modules.
- In object oriented development the graph may be called a state chart .

- STG/state charts are commonly depicted by a set of nodes (circles, ovals, rounded rectangles) which represent states.
- These usually will have a name or number to identify the state.
- A set of arrows between nodes indicate what inputs or events will cause a transition or change between the two linked states.
- Outputs/actions occurring with a state transition are also depicted on a link or arrow



- From the tester's view point the review should ensure that
- (i) the proper number of states are represented,
- (ii) each state transition (input/output/action) is correct,
- (iii) equivalent states are identified,
- (iv) unreachable and dead states are identified.

For the simple state machine in Figure transitions to be tested are:

- Input A in S1
- Input A in S2
- Input B in S1
- Input B in S2
- Input C in S1
- Input C in S2

- **Error Guessing**
  - Designing test cases using the error guessing approach is based on the tester's/developer's past experience with code similar to the code-under test, and their intuition as to where defects may lurk in the code.
  - Code similarities may extend to the structure of the code, its domain, the design approach used, its complexity, and other factors.
  - The tester/developer is sometimes able to make an educated "guess" as to which types of defects may be present and design test cases to reveal them.

- Some examples of obvious types of defects to test for are cases where there is a possible division by zero, where there are a number of pointers that are manipulated, or conditions around array boundaries
- Error guessing is an ad hoc approach to test design in most cases.
- However, if defect data for similar code or past releases of the code has been carefully recorded, the defect types classified, and failure symptoms due to the defects carefully noted, this approach can have some structure and value.

Compatibility Testing

- Software compatibility testing means checking that your software interacts with and shares information correctly with other software.
- This interaction could occur between two programs simultaneously running on the same computer or even on different computers connected through the Internet thousands of miles apart.
- The interaction could also be as simple as saving data to a floppy disk and hand-carrying it to another computer across the room.

Example of Compatible Software

- □ Cutting text from a web page and pasting it into a document opened in your word processor
- □ Saving accounting data from one spreadsheet program and then loading it into a completely different spreadsheet program
- □ λ Having photograph touchup software work correctly on different versions of the same operating system
- □ Having your word processor load in the names and addresses from your contact management program and print out personalized invitations and envelopes
- □ λ Upgrading to a new database program and having all your existing databases load in and work just as they did with the old program

Performing Software Compatibility

Testing

λ Software compatibility testing on a new piece of software, need to get the answers to a few questions:

- What other platforms (operating system, web browser, or other operating environment) and other application software is your software designed to be compatible with?
- If the software you're testing is a platform, what applications are designed to run under it?
- What compatibility standards or guidelines should be followed that define how your software should interact with other software?
- What types of data will your software use to interact and share information with other platforms and software?

Performing Software Compatibility

Testing

- ❑ Gaining the answers to these questions is basic static testing both black-box and white-box.
- ❑ It involves thoroughly analyzing the specification for the product and any supporting specifications.
- ❑ It could also entail discussions with the programmers and possibly close review of the code to assure that all links to and from your software are identified.

Platform and Application Versions

- ❑ Selecting the target platforms or the compatible applications is really a program management or a marketing task.
- ❑ Each platform has its own development criteria and it's important, from a project management standpoint, to make this platform list as small as possible but still fill the customer's needs.

Backward and Forward

Compatibility

❑Two terms regarding compatibility testing are backward compatible and forward compatible.

❑If something is backward compatible, it will work with previous versions of the software.

❑If something is forward compatible, it will work with future versions of the software.

The Impact of Testing Multiple

Versions

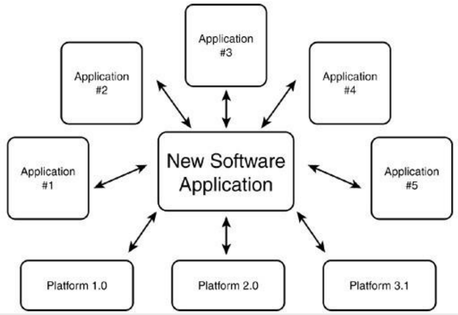
- ❑ Testing that multiple versions of platforms and software applications work properly with each other can be a huge task.
- ❑ Consider the situation of having to compatibility test a new version of a popular operating system.
- ❑ The programmers have made numerous bug fixes and performance improvements and have added many new features to the code.
- ❑ There could be tens or hundreds of thousands of existing programs for the current versions of the OS. The project's goal is to be 100 percent compatible with them.

Compatibility Testing of new application

The key word is **important**. The criteria that might go into deciding what programs to choose could be:

- ❑ – **Popularity**. Use sales data to select the top 100 or 1,000 most popular programs.
- ❑ **Age**. You might want to select programs and versions that are less than three years old.
- ❑ **Type**. Break the software world into types such as painting, writing, accounting, databases, communications, and so on. Select software from each category for testing.
- ❑ **Manufacturer**. Another criteria would be to pick software based on the company that created it.

Compatibility Testing of new application



Standards and Guidelines

- ❑ There are two levels of requirements:
- ❑ **high-level:** High-level standards are the ones that guide your product's general operation, Its look and feel, its supported features, and so on.
- ❑ **Low-level:** Low-level standards are the nitty-gritty details, such as the file formats and the network communications protocols.
- ❑ Both are important and both need to be tested to assure compatibility.

High-Level Standards and

❑ **Guidelines**

- ❑ Will your software run under Windows, Mac, or Linux operating systems?
- ❑ Is it a web application? If so, what browsers will it run on?
- ❑ Each of these is considered a platform and most have their own set of standards and guidelines that must be followed if an application is to claim that it's compatible with the platform.

Low-Level Standards and

Guidelines

- ❑ Low-level standards are more important than the high-level standards.
- ❑ creating a program that would run on Windows that didn't have the look and feel of other Windows software.
- ❑ It wouldn't be granted the Certified for Microsoft Windows logo.
- ❑ Users might not be thrilled with the differences from other applications, but they could use the product.

❑ **Low Level Standard - Guidelines**

- ❑ Software is a graphics program that saves its files to disk as .pict files (a standard Macintosh file format for graphics) but the program doesn't follow the standard for .pict files, users won't be able to view the files in any other program.
- ❑ Software wouldn't be compatible with the standard and would likely be a short-lived product.

Guidelines

- ❑ Communications protocols, programming language syntax, and any means that programs use to share information must adhere to published standards and guidelines.
- ❑ These low-level standards are often taken for granted It must be tested from a tester's perspective
- ❑ Treat low-level compatibility standards as an extension of the software's specification
- ❑ For example if software spec states, "The software will save and load its graphics files as .bmp, .jpg, and .gif formats,"
- ❑ Need to find the standards for these formats and design tests to confirm that the software does indeed adhere to them.

Data Sharing Compatibility

- ❑ The sharing of data among applications is really gives software its power.
- ❑ A well-written program that supports and adheres to published standards and allows users to easily transfer data to and from other software is a great compatible product.
- ❑ The most familiar means of transferring data from one program to another is saving and loading disk files.

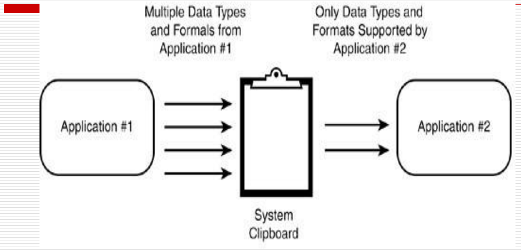
Data Sharing Compatibility λ few examples:

- ❑ – File save and file load are the data-sharing methods.
- ❑ The data format of the files needs to meet standards for it to be compatible on multiple computers.
- ❑ File export and file import are the means that many programs use to be compatible with older versions of themselves and with other programs.

–

Data Sharing Compatibility

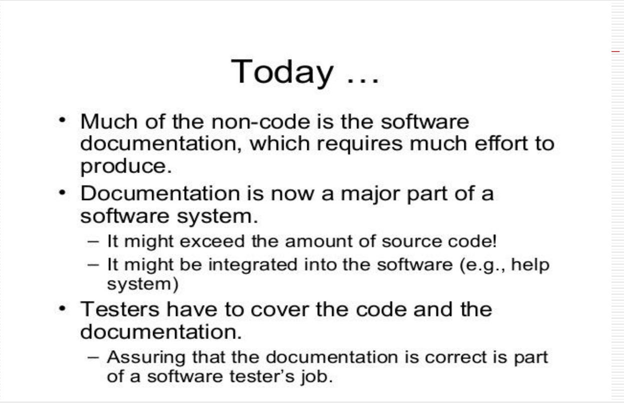
- ❑ Cut, copy, and paste are the most familiar methods for sharing data among programs without transferring the data to a disk.


Multiple Data Types and Formats from Application #1
Only Data Types and Formats Supported by Application #2
Application #1
Application #2
System Clipboard

Data Sharing Compatibility

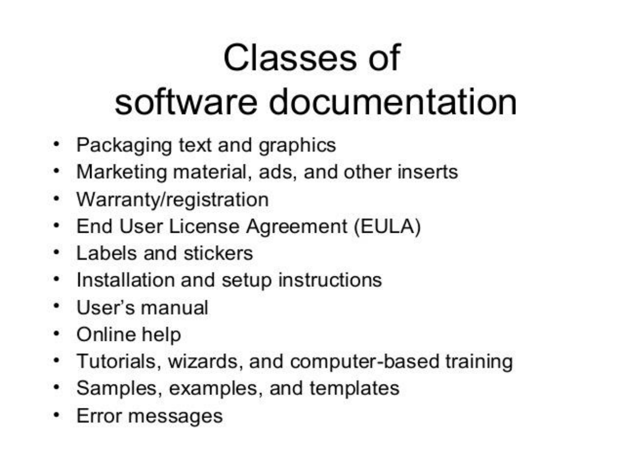
- ❑ The Clipboard is designed to hold several different data types.
- ❑ DDE (pronounced D-D-E), COM (for Component Object Model), and OLE (pronounced oh-lay) are the methods in Windows of transferring data between two applications. DDE stands for Dynamic Data Exchange and OLE stands for Object Linking and Embedding.
- ❑ Other platforms support similar methods.

User Documentation Testing



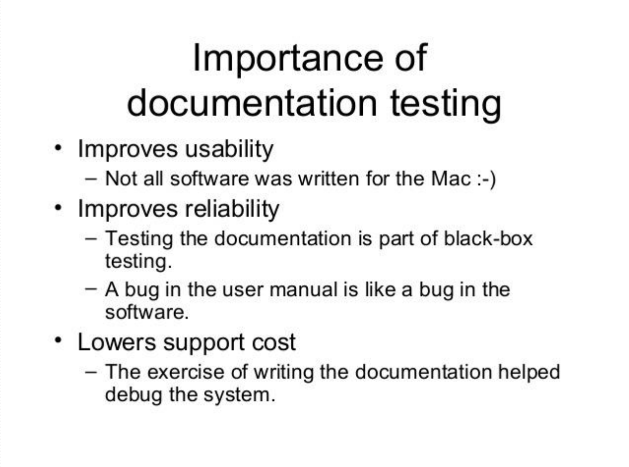
Today …

- Much of the non-code is the software documentation, which requires much effort to produce.
- Documentation is now a major part of a software system.
  - It might exceed the amount of source code!
  - It might be integrated into the software (e.g., help system)
- Testers have to cover the code and the documentation.
  - Assuring that the documentation is correct is part of a software tester's job.



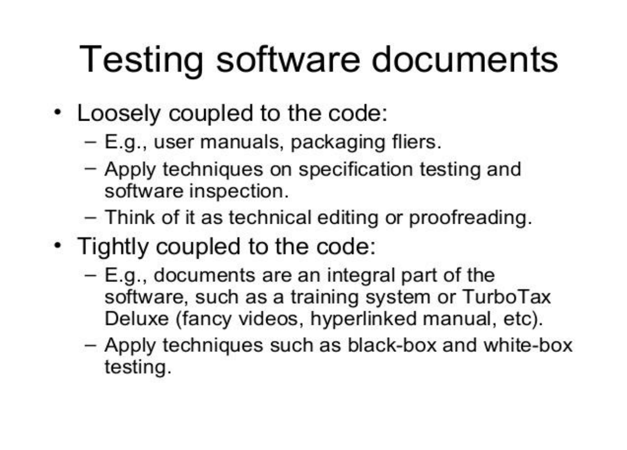
Classes of software documentation

- Packaging text and graphics
- Marketing material, ads, and other inserts
- Warranty/registration
- End User License Agreement (EULA)
- Labels and stickers
- Installation and setup instructions
- User's manual
- Online help
- Tutorials, wizards, and computer-based training
- Samples, examples, and templates
- Error messages



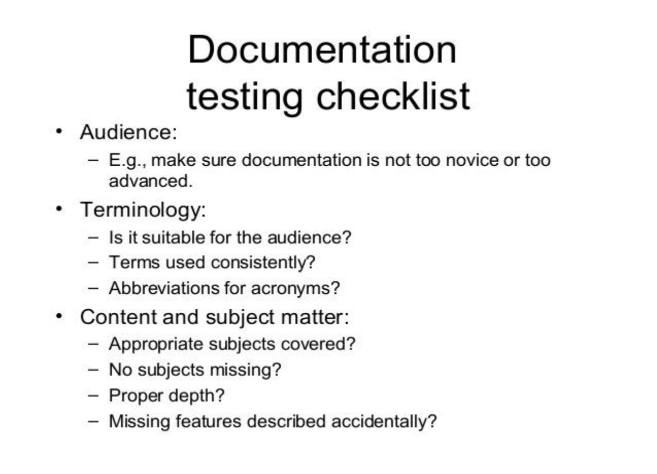
Importance of documentation testing

- Improves usability
  - Not all software was written for the Mac :-)
- Improves reliability
  - Testing the documentation is part of black-box testing.
  - A bug in the user manual is like a bug in the software.
- Lowers support cost
  - The exercise of writing the documentation helped debug the system.



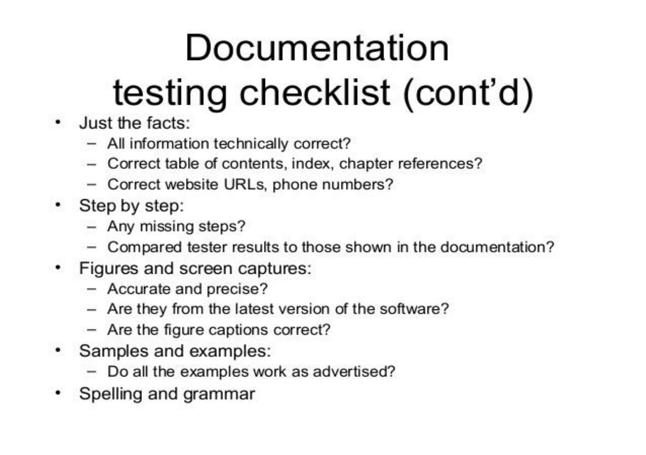
Testing software documents

- Loosely coupled to the code:
  - E.g., user manuals, packaging fliers.
  - Apply techniques on specification testing and software inspection.
  - Think of it as technical editing or proofreading.
- Tightly coupled to the code:
  - E.g., documents are an integral part of the software, such as a training system or TurboTax Deluxe (fancy videos, hyperlinked manual, etc).
  - Apply techniques such as black-box and white-box testing.



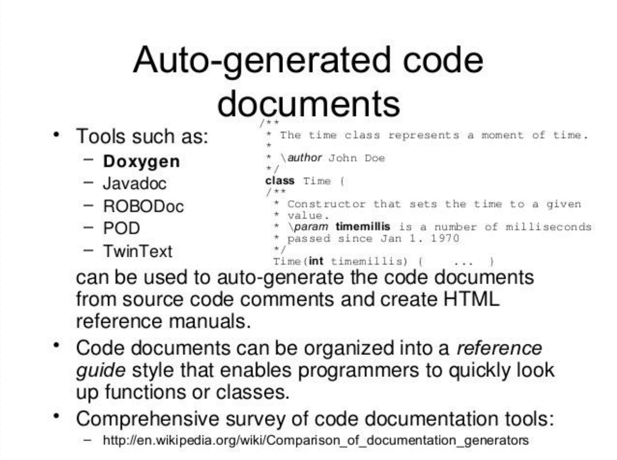
Documentation testing checklist

- Audience:
  - E.g., make sure documentation is not too novice or too advanced.
- Terminology:
  - Is it suitable for the audience?
  - Terms used consistently?
  - Abbreviations for acronyms?
- Content and subject matter:
  - Appropriate subjects covered?
  - No subjects missing?
  - Proper depth?
  - Missing features described accidentally?



Documentation testing checklist (cont'd)

- Just the facts:
  - All information technically correct?
  - Correct table of contents, index, chapter references?
  - Correct website URLs, phone numbers?
- Step by step:
  - Any missing steps?
  - Compared tester results to those shown in the documentation?
- Figures and screen captures:
  - Accurate and precise?
  - Are they from the latest version of the software?
  - Are the figure captions correct?
- Samples and examples:
  - Do all the examples work as advertised?
- Spelling and grammar



Auto-generated code documents

- Tools such as:
  - **Doxygen**
  - Javadoc
  - ROBODoc
  - POD
  - TwinText

```
/**
 * The time class represents a moment of time.
 *
 * \author John Doe
 */
class Time {
 /**
  * Constructor that sets the time to a given
  * value.
  * \param timemillis is a number of milliseconds
  * passed since Jan 1. 1970
  */
  Time(int timemillis) {    ...  }
```

can be used to auto-generate the code documents from source code comments and create HTML reference manuals.

- Code documents can be organized into a *reference guide* style that enables programmers to quickly look up functions or classes.
- Comprehensive survey of code documentation tools:
  - http://en.wikipedia.org/wiki/Comparison_of_documentation_generators

Domain Testing

- ❑ Domain Testing is a type of Functional Testing which tests the application by giving inputs and evaluating its appropriate outputs.
- ❑ It is a software testing technique in which the output of a system has to be tested with a minimum number of inputs in such a case to ensure that the system does not accept invalid and out of range input values.

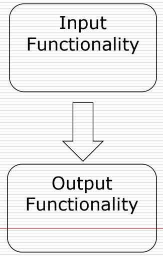
- □ One of the most important White Box Testing method is a domain testing.
- □ The main goal of the Domain testing is to check whether the system accepts the input within the acceptable range and delivers the required output.
- □ Also, it verifies the system should not accept the inputs, conditions and indices outside the specified or valid range.

- □ Domain testing is different from domain specific knowledge you need to test a software system.
- □ In this tutorial, you will learn-
- □ Simpler Practice of Domain Testing
- □ Domain Testing Strategy
- □ Domain Testing Example
- □ Domain Testing Structure

- In domain testing, we divide a domain into sub-domains (equivalence classes) and then test using values from each subdomain. For example, if a website (domain) has been given for testing, we will be dividing the website into small portions (subdomain) for the ease of testing.
- Practitioners often study the simplest cases of domain testing less than two other names, "boundary testing" and "equivalence class analysis.“

- **Domain Testing Strategy**
- While domain testing, you need to consider following things,
  - What domain are we testing?
  - How to group the values into classes?
  - Which values of the classes to be tested?
  - How to determine the result?

- **What domain are we testing?**
- Any domain which we test has some input functionality and an output functionality. There will be some input variables to be entered, and the appropriate output has to be verified.



Input Functionality

Output Functionality

- **Domain Testing Structure**
- Usually, testers follow the below steps in a domain testing. These may be customized/ skipped according to our testing needs.
  - Identify the potentially interesting variables.
  - Identify the variable(s) you can analyze now and order them (smallest to largest and vice versa).
  - Create and identify boundary values and equivalence class values.
  - Identify and test variables that hold results (output variables).
  - Evaluate how the program uses the value of this variable.