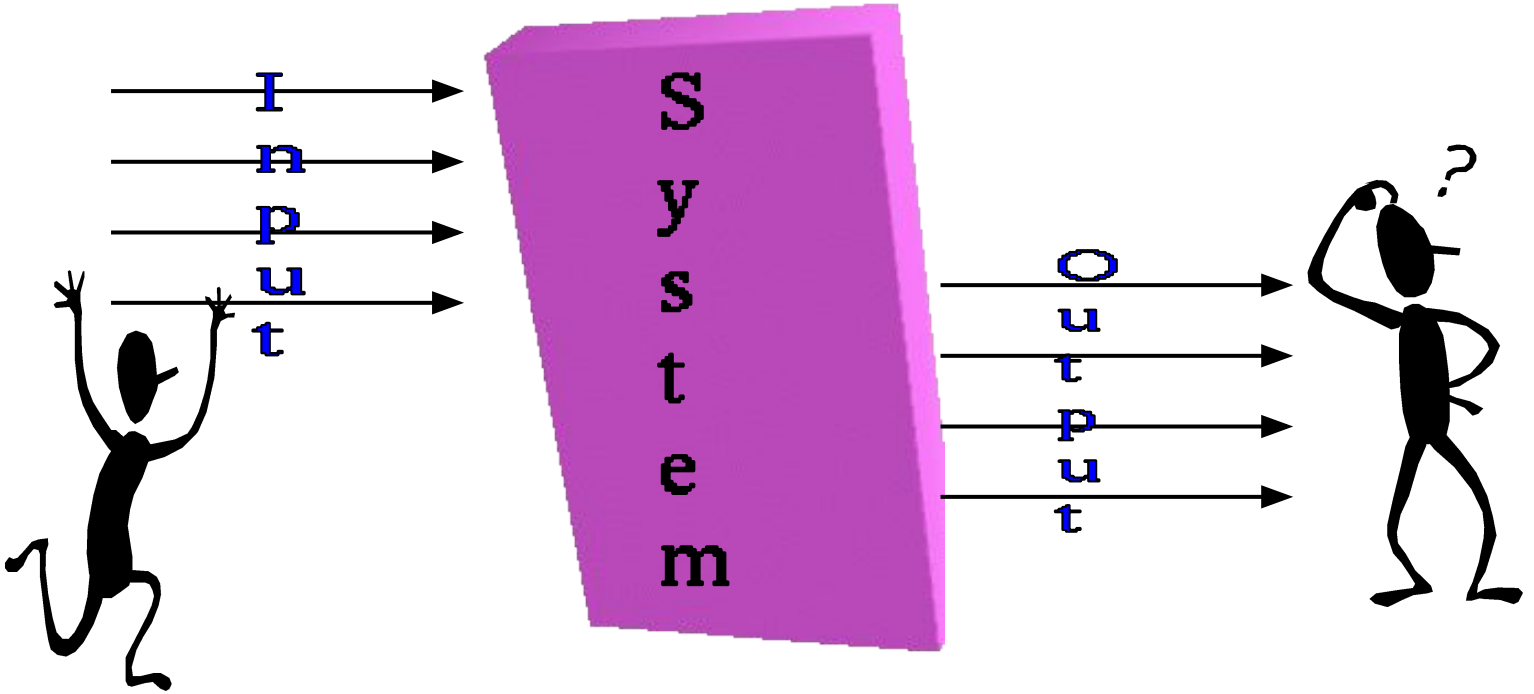
# Testing

## (Unit 5)

# How Do You Test a Program?

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

# How Do You Test a Program?



# How Do You Test a Program?

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

# Important Terminologies

Mistake, Error, Failure, Test case, Test scenario, Test script, Test suite, Testability, Failure mode, Equivalent faults

- **Mistake**: It is essentially any programmer action that later shows up as an incorrect result during program execution. E.g, uninitialization of some variables, overlooking exceptional conditions like division by zero. It can lead to an incorrect result.
- **Error:** It is the result of a mistake committed by a developer in any of the development activities. E.g an error is call made to a wrong function.

- **Failure** : A failure is a manifestation of an error (aka defect or bug).
  - Mere presence of an error may not lead to a failure.

  E.g A program crashes on an input, A robot fails to avoid an obstacle and collides with it.

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

  Example

- **Positive Test Case:** It is designed to test whether the software correctly performs a required functionality.
- **Negative Test case:** It is designed to test whether the software carries out something that is not required of the system.

**Example: Login related test case**

- **Test scenario:** It is an abstract test case as it only identifies the aspects of the program that are to be tested without identifying the input, state or output.

The test case is an implementation of a test scenario.

- **Test Script :** It is an encoding of a test case as a short program. Developed for automated execution of test cases.

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

- **Testability:** testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.
- **Failure Mode:** of a software denotes an observable way in which it can fail. E.g. for a online train reservation system, the failure modes are: failing to book an available seat, incorrect seat booking and system crash.

- **Equivalent Fault:** It denote two or more bugs that result in the system failing in the same failure mode.
- E.g. division by zero and illegal memory access errors leads to program crash.

# Verification versus Validation

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

# Verification versus Validation

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

# Overview of Testing Activities

- Test Suite Design
- Run test cases and observe results to detect failures.
- Debug to locate errors
- Correct errors.

# Testing process

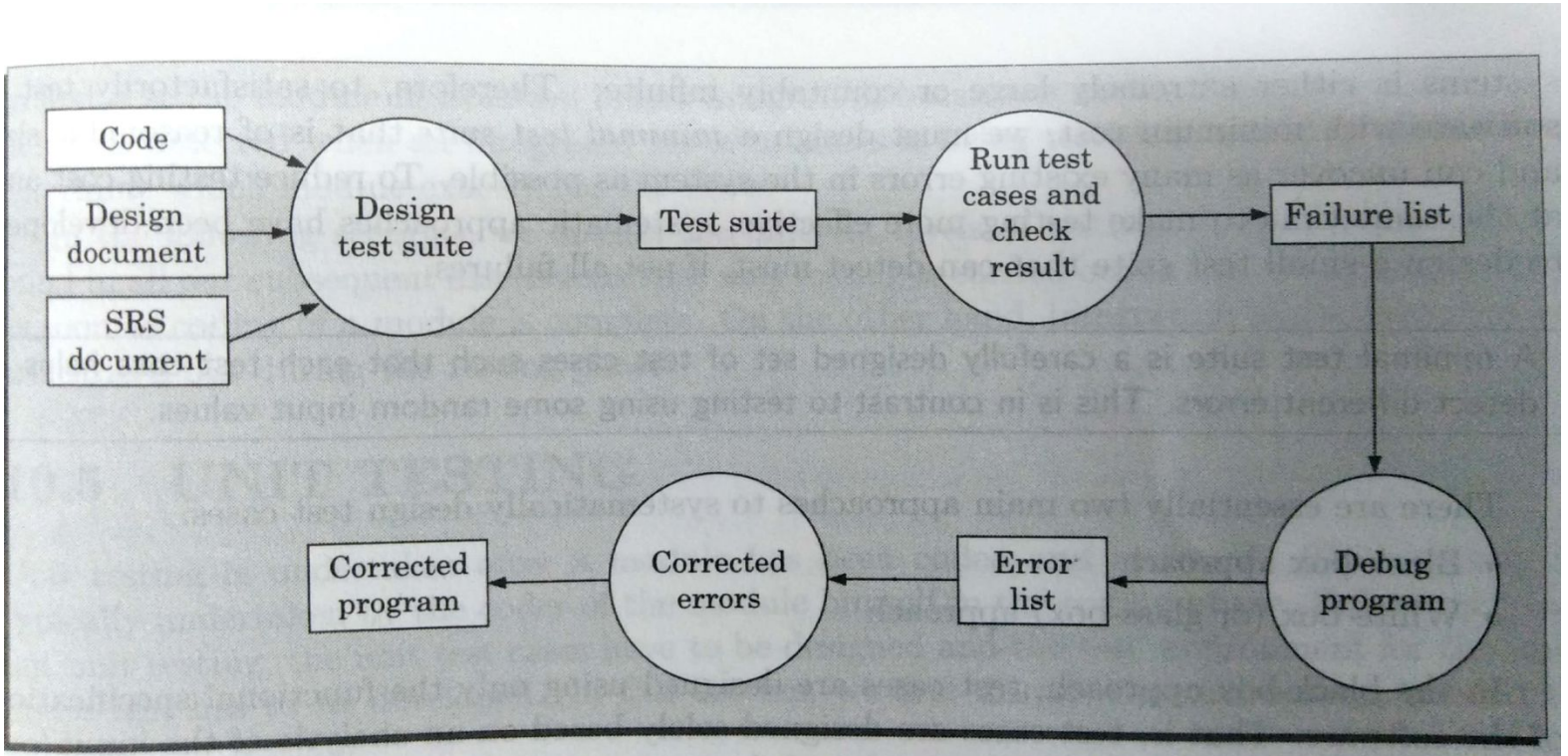



**Figure 10.2:** Testing process.

# Design of Test Cases

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

# Design of Test Cases

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

# Design of Test Cases

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

# Design of Test Cases

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

# Design of Test Cases

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

# Design of Test Cases

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

# Black-Box Testing

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

# White-box Testing

- Designing white-box test cases:
  - Requires knowledge about the internal structure of software.
  - White-box testing is also called structural testing.
  - In this unit we will not study white-box testing.

# Black-Box Testing

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

# Equivalence Class Partitioning

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

# Why Define Equivalence Classes?

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

# Equivalence Class Partitioning

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

# Equivalence Class Partitioning

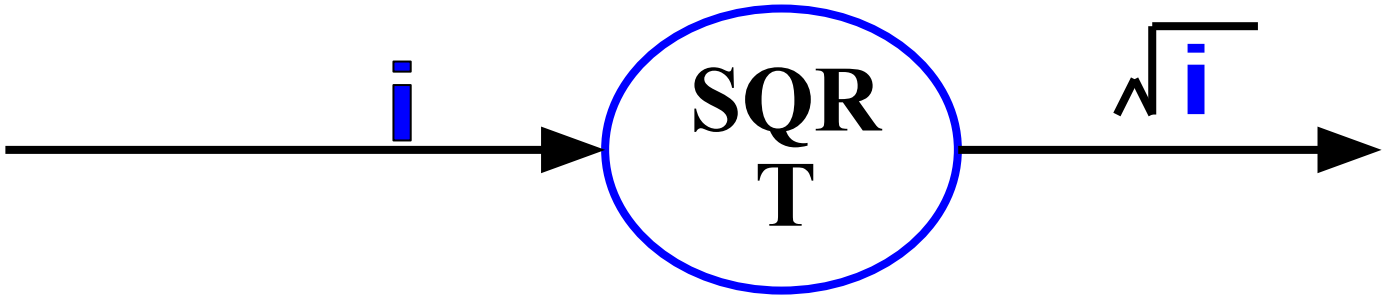
- If the input data to the program is specified by a **range of values:**
  - e.g. numbers between 1 to 5000.
  - One valid and two invalid equivalence classes are defined.

# Equivalence Class Partitioning

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

# Example

- A program reads an input value in the range of 1 and 5000:
  - Computes the square root of the input number

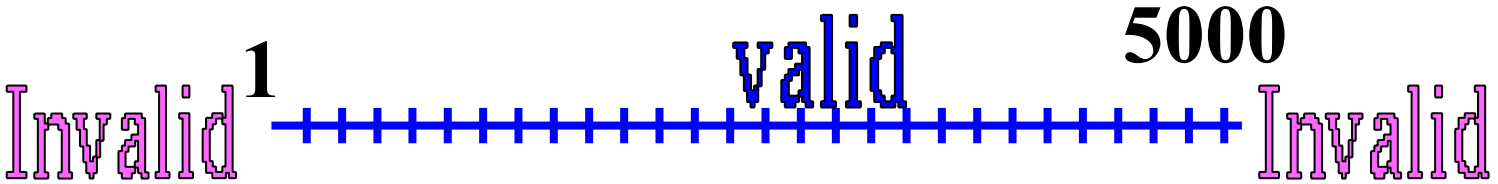


# Example (cont.)

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

# Example (cont.)

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



# Boundary Value Analysis

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

# Boundary Value Analysis

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

# Example

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

# Examples

**Example- 8.I**

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.



39

**Solution**

Quadratic equation will be of type:

$ax^2+bx+c=0$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$



40

# Software Testing

The boundary value test cases are :

| Test Case | a | b | c | Expected output |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not Quadratic |
| 2 | 1 | 50 | 50 | Real Roots |
| 3 | 50 | 50 | 50 | Imaginary Roots |
| 4 | 99 | 50 | 50 | Imaginary Roots |
| 5 | 100 | 50 | 50 | Imaginary Roots |
| 6 | 50 | 0 | 50 | Imaginary Roots |
| 7 | 50 | 1 | 50 | Imaginary Roots |
| 8 | 50 | 99 | 50 | Imaginary Roots |
| 9 | 50 | 100 | 50 | Equal Roots |
| 10 | 50 | 50 | 0 | Real Roots |
| 11 | 50 | 50 | 1 | Real Roots |
| 12 | 50 | 50 | 99 | Imaginary Roots |
| 13 | 50 | 50 | 100 | Imaginary Roots |

**Example – 8.2**

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

$1900 \leq \text{year} \leq 2025$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

**Solution**

The Previous date program takes a date as input and checks it for validity. If valid, it returns the previous date as its output.

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13.



43

# Software Testing

The boundary value test cases are:

| Test Case | Month | Day | Year | Expected output |
|---|---|---|---|---|
| 1 | 6 | 15 | 1900 | 14 June, 1900 |
| 2 | 6 | 15 | 1901 | 14 June, 1901 |
| 3 | 6 | 15 | 1962 | 14 June, 1962 |
| 4 | 6 | 15 | 2024 | 14 June, 2024 |
| 5 | 6 | 15 | 2025 | 14 June, 2025 |
| 6 | 6 | 1 | 1962 | 31 May, 1962 |
| 7 | 6 | 2 | 1962 | 1 June, 1962 |
| 8 | 6 | 30 | 1962 | 29 June, 1962 |
| 9 | 6 | 31 | 1962 | Invalid date |
| 10 | 1 | 15 | 1962 | 14 January, 1962 |
| 11 | 2 | 15 | 1962 | 14 February, 1962 |
| 12 | 11 | 15 | 1962 | 14 November, 1962 |
| 13 | 12 | 15 | 1962 | 14 December, 1962 |



44

**Example – 8.3**

Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle]

Design the boundary value test cases.

## Solution

The boundary value test cases are shown below:

| Test case | x | y | z | Expected Output |
|---|---|---|---|---|
| 1 | 50 | 50 | 1 | Isosceles |
| 2 | 50 | 50 | 2 | Isosceles |
| 3 | 50 | 50 | 50 | Equilateral |
| 4 | 50 | 50 | 99 | Isosceles |
| 5 | 50 | 50 | 100 | Not a triangle |
| 6 | 50 | 1 | 50 | Isosceles |
| 7 | 50 | 2 | 50 | Isosceles |
| 8 | 50 | 99 | 50 | Isosceles |
| 9 | 50 | 100 | 50 | Not a triangle |
| 10 | 1 | 50 | 50 | Isosceles |
| 11 | 2 | 50 | 50 | Isosceles |
| 12 | 99 | 50 | 50 | Isosceles |
| 13 | 100 | 50 | 50 | Not a triangle |
| | | | | |



46

## Example 8.7

Consider the program for the determination of nature of roots of a quadratic equation as explained in example 8.1. Identify the equivalence class test cases for output and input domains.



47

## Solution

Output domain equivalence class test cases can be identified as follows:

$O_1$={<a,b,c>:Not a quadratic equation if a = 0}
$O_1$={<a,b,c>:Real roots if ($b^2$-4ac)>0}
$O_1$={<a,b,c>:Imaginary roots if ($b^2$-4ac)<0}
$O_1$={<a,b,c>:Equal roots if ($b^2$-4ac)=0}`

The number of test cases can be derived form above relations and shown below:

| *Test case* | *a* | *b* | *c* | *Expected output* |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not a quadratic equation |
| 2 | 1 | 50 | 50 | Real roots |
| 3 | 50 | 50 | 50 | Imaginary roots |
| 4 | 50 | 100 | 50 | Equal roots |



48

# *Software Testing*

We may have another set of test cases based on input domain.

$I_1 = \{a: a = 0\}$

$I_2 = \{a: a < 0\}$

$I_3 = \{a: 1 \leq a \leq 100\}$

$I_4 = \{a: a > 100\}$

$I_5 = \{b: 0 \leq b \leq 100\}$

$I_6 = \{b: b \leq 0\}$

$I_7 = \{b: b > 100\}$

$I_8 = \{c: 0 \leq c \leq 100\}$

$I_9 = \{c: c < 0\}$

$I_{10} = \{c: c > 100\}$



49

| Test Case | a | b | c | Expected output |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not a quadratic equation |
| 2 | -1 | 50 | 50 | Invalid input |
| 3 | 50 | 50 | 50 | Imaginary Roots |
| 4 | 101 | 50 | 50 | invalid input |
| 5 | 50 | 50 | 50 | Imaginary Roots |
| 6 | 50 | -1 | 50 | invalid input |
| 7 | 50 | 101 | 50 | invalid input |
| 8 | 50 | 50 | 50 | Imaginary Roots |
| 9 | 50 | 50 | -1 | invalid input |
| 10 | 50 | 50 | 101 | invalid input |

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.



50

## Example 8.8

Consider the program for determining the previous date in a calendar as explained in example 8.3. Identify the equivalence class test cases for output & input domains.



51

## Solution

Output domain equivalence class are:

$O_1$={<D,M,Y>: Previous date if all are valid inputs}
$O_1$={<D,M,Y>: Invalid date if any input makes the date invalid}

| Test case | M | D | Y | Expected output |
|---|---|---|---|---|
| 1 | 6 | 15 | 1962 | 14 June, 1962 |
| 2 | 6 | 31 | 1962 | Invalid date |

We may have another set of test cases which are based on input domain.

$I_1$={month: $1 \leq m \leq 12$}

$I_2$={month: $m < 1$}

$I_3$={month: $m > 12$}

$I_4$={day: $1 \leq D \leq 31$}

$I_5$={day: $D < 1$}

$I_6$={day: $D > 31$}

$I_7$={year: $1900 \leq Y \leq 2025$}

$I_8$={year: $Y < 1900$}

$I_9$={year: $Y > 2025$}



53

Inputs domain test cases are :

| Test Case | M | D | Y | Expected output |
|---|---|---|---|---|
| 1 | 6 | 15 | 1962 | 14 June, 1962 |
| 2 | -1 | 15 | 1962 | Invalid input |
| 3 | 13 | 15 | 1962 | invalid input |
| 4 | 6 | 15 | 1962 | 14 June, 1962 |
| 5 | 6 | -1 | 1962 | invalid input |
| 6 | 6 | 32 | 1962 | invalid input |
| 7 | 6 | 15 | 1962 | 14 June, 1962 |
| 8 | 6 | 15 | 1899 | invalid input (Value out of range) |
| 9 | 6 | 15 | 2026 | invalid input (Value out of range) |



54

## Example – 8.9

Consider the triangle problem specified in a example 8.3. Identify the equivalence class test cases for output and input domain.



55

## Solution

Output domain equivalence classes are:

$O_1$={<x,y,z>: Equilateral triangle with sides x,y,z}
$O_1$={<x,y,z>: Isosceles triangle with sides x,y,z}
$O_1$={<x,y,z>: Scalene triangle with sides x,y,z}
$O_1$={<x,y,z>: Not a triangle with sides x,y,z}

The test cases are:

| Test case | x | y | z | Expected Output |
|---|---|---|---|---|
| 1 | 50 | 50 | 50 | Equilateral |
| 2 | 50 | 50 | 99 | Isosceles |
| 3 | 100 | 99 | 50 | Scalene |
| 4 | 50 | 100 | 50 | Not a triangle |

Input domain based classes are:

$I_1 = \{x: x < 1\}$

$I_2 = \{x: x > 100\}$

$I_3 \equiv \{x: 1 \leq x \leq 100\}$

$I_4 = \{y: y < 1\}$

$I_5 = \{y: y > 100\}$

$I_6 \equiv \{y: 1 \leq y \leq 100\}$

$I_7 = \{z: z < 1\}$

$I_8 = \{z: z > 100\}$

$I_9 = \{z: 1 \leq z \leq 100\}$



57

# *Software Testing*

Some inputs domain test cases can be obtained using the relationship amongst x,y and z.

$I_{10}=\{< x,y,z >: x = y = z\}$

$I_{11}=\{< x,y,z >: x = y,\ x \neq z\}$

$I_{12}=\{< x,y,z >: x = z,\ x \neq y\}$

$I_{13}=\{< x,y,z >: y \neq z,\ x \neq y,\ x \neq z\}$

$I_{14}=\{< x,y,z >: x \neq y,\ x \neq z,\ y \neq z\}$

$I_{15}=\{< x,y,z >: x = y + z\}$

$I_{16}=\{< x,y,z >: x > y + z\}$

$I_{17}=\{< x,y,z >: y = x + z\}$

$I_{18}=\{< x,y,z >: y > x + z\}$

$I_{19}=\{< x,y,z >: z = x + y\}$

$I_{20}=\{< x,y,z >: z > x + y\}$



58

## Test cases derived from input domain are:

| Test case | x | y | z | Expected Output |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Invalid input |
| 2 | 101 | 50 | 50 | Invalid input |
| 3 | 50 | 50 | 50 | Equilateral |
| 4 | 50 | 0 | 50 | Invalid input |
| 5 | 50 | 101 | 50 | Invalid input |
| 6 | 50 | 50 | 50 | Equilateral |
| 7 | 50 | 50 | 0 | Invalid input |
| 8 | 50 | 50 | 101 | Invalid input |
| 9 | 50 | 50 | 50 | Equilateral |
| 10 | 60 | 60 | 60 | Equilateral |
| 11 | 50 | 50 | 60 | Isosceles |
| 12 | 50 | 60 | 50 | Isosceles |
| 13 | 60 | 50 | 50 | Isosceles |
| | | | | |

(Contd.)...

# Software Testing

| Test case | x | y | z | Expected Output |
|---|---|---|---|---|
| 14 | 100 | 99 | 50 | Scalene |
| 15 | 100 | 50 | 50 | Not a triangle |
| 16 | 100 | 50 | 25 | Not a triangle |
| 17 | 50 | 100 | 50 | Not a triangle |
| 18 | 50 | 100 | 25 | Not a triangle |
| 19 | 50 | 50 | 100 | Not a triangle |
| 20 | 25 | 50 | 100 | Not a triangle |
| | | | | |



60

# Debugging

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

# Brute-Force method

- This is the most common method of debugging:
  - Least efficient method.
  - Program is loaded with print statements
  - Print the intermediate values
  - Hope that some of printed values will help identify the error.

# Symbolic Debugger

- Brute force approach becomes more systematic:
  - With the use of a symbolic debugger,
  - Symbolic debuggers get their name for historical reasons
  - Early debuggers let you only see values from a program dump:
    - Determine which variable it corresponds to.

# Symbolic Debugger

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

# Backtracking

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

# Backtracking

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

# Cause-elimination method

- Determine a list of causes:
  - which could possibly have contributed to the error symptom.
  - tests are conducted to eliminate each.
- A related technique of identifying error by examining error symptoms:
  - software fault tree analysis.

# Program Slicing

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

# Example

```
int main(){
 int i,s;
 i=1; s=1;
 while(i<=10){
    s=s+i;
    i++;}
printf("%d",s);
printf("%d",i);
}
```

# Debugging Guidelines

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

# Debugging Guidelines

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

# Summary

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

# Summary

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

# Summary

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

# Summary

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