Software Testing

Dr. Durga Prasad Mohapatra
Professor
Department Of Computer Science & Engineering
NIT Rourkela

Software Development Process

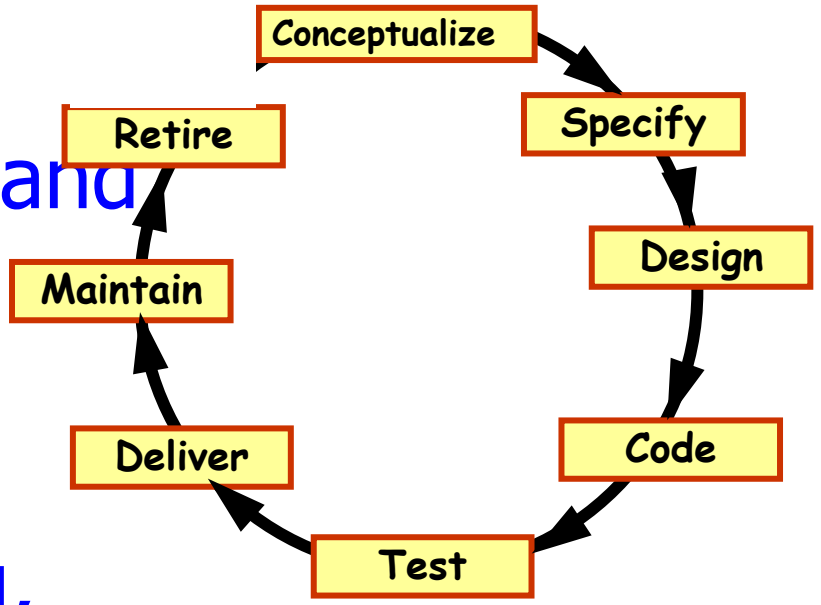
Software Development Life Cycle (or software development process):

- Series of identifiable stages that a software product undergoes during its life time:
  - Feasibility study
  - Requirements analysis and specification,
  - Design,
  - Coding,
  - Testing
  - Maintenance.

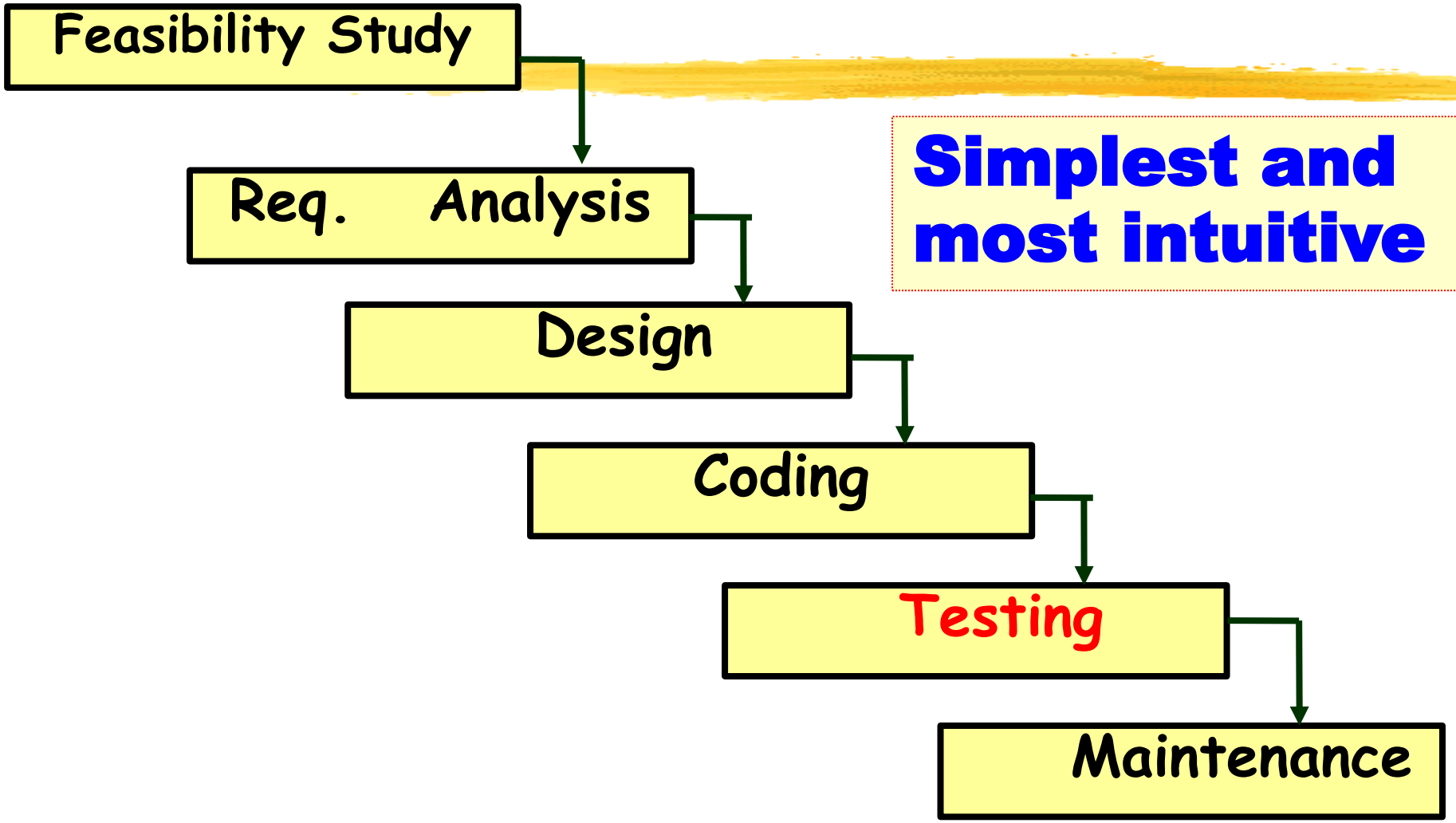
Software Life Cycle

Classical Waterfall Model

- Classical waterfall model divides life cycle into following phases:
  - Feasibility study,
  - Requirements analysis and specification,
  - Design,
  - Coding and unit testing,
  - Integration and system testing,
  - Maintenance.



Classical Waterfall Model



Simplest and most intuitive

Defect Reduction Techniques

- Review
- Testing
- Formal verification
- Development process
- Systematic methodologies

Why to Test?

- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
  - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
    - The floating point number was larger than 32767
    - Efficiency considerations had led to the disabling of the exception handler.
- Total Cost: over $1 billion

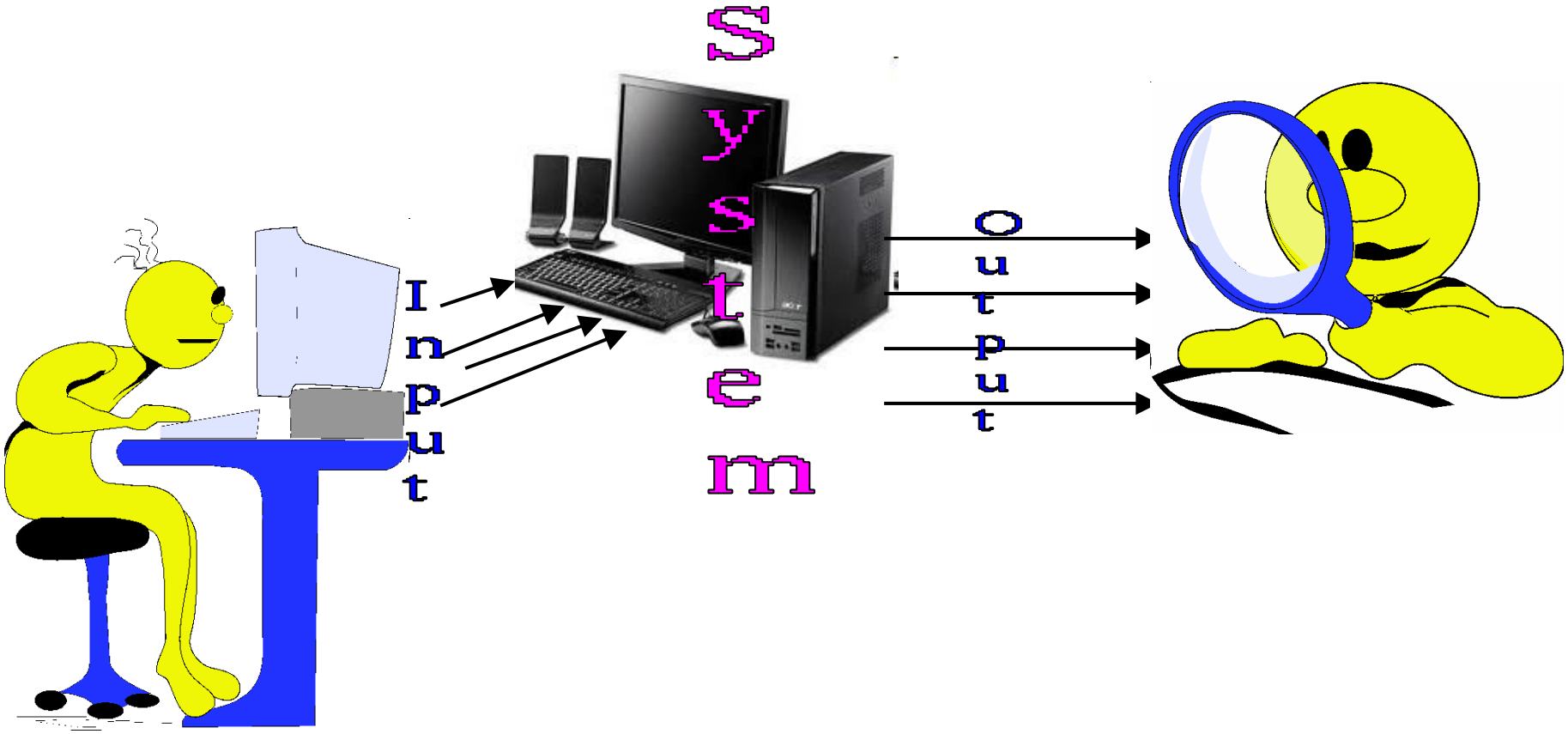
Organization of this lecture

- Important concepts in program testing
- Black-box testing:
  - equivalence partitioning
  - boundary value analysis
- White-box testing
- Debugging
- Unit, Integration, and System testing
- Summary

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.

What's So Hard About Testing ?

- Consider `int proc1(int x, int y)`
- Assuming a 64 bit computer
  - Input space = 2^{128}
- Assuming it takes 10secs to key-in an integer pair
  - It would take about a billion years to enter all possible values!
  - Automatic testing has its own problems!

Testing Facts

- Consumes largest effort among all phases
  - Largest manpower among all other development roles
  - Implies more job opportunities
- About 50% development effort
  - But 10% of development time?
  - How?

Testing Facts

- Testing is getting more complex and sophisticated every year.
  - Larger and more complex programs
  - Newer programming paradigms

Overview of Testing Activities

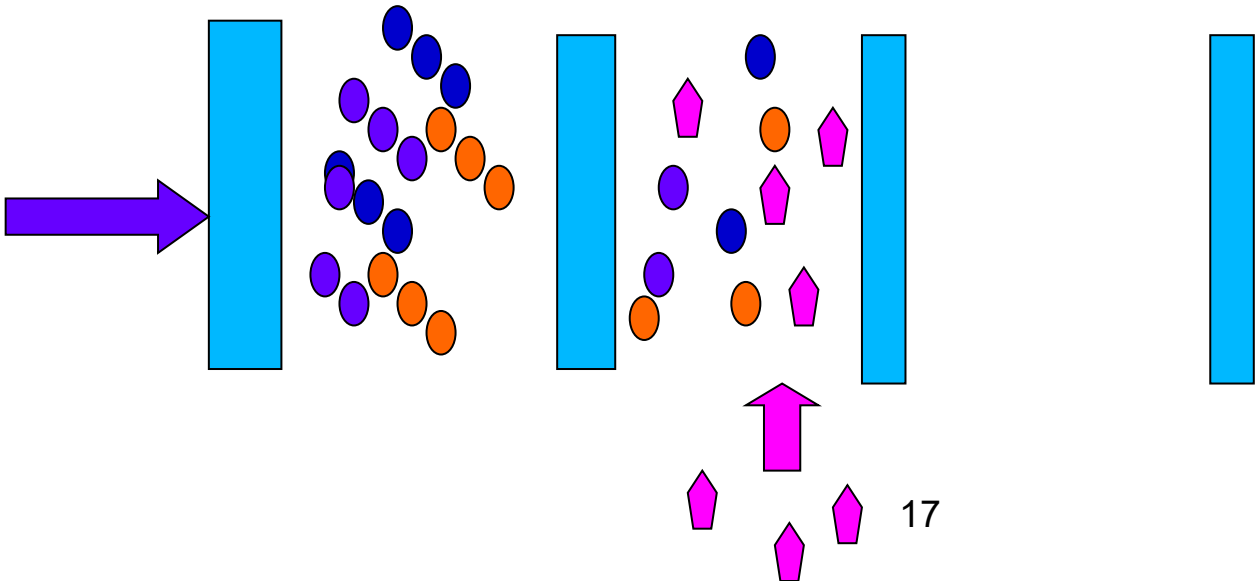
- Test Suite Design
- Run test cases and observe results to detect failures.
- Debug to locate errors
- Correct errors.

Error, Faults, and Failures

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

Pesticide Effect

- Errors that escape a fault detection technique:
  - Can not be detected by further applications of that technique.



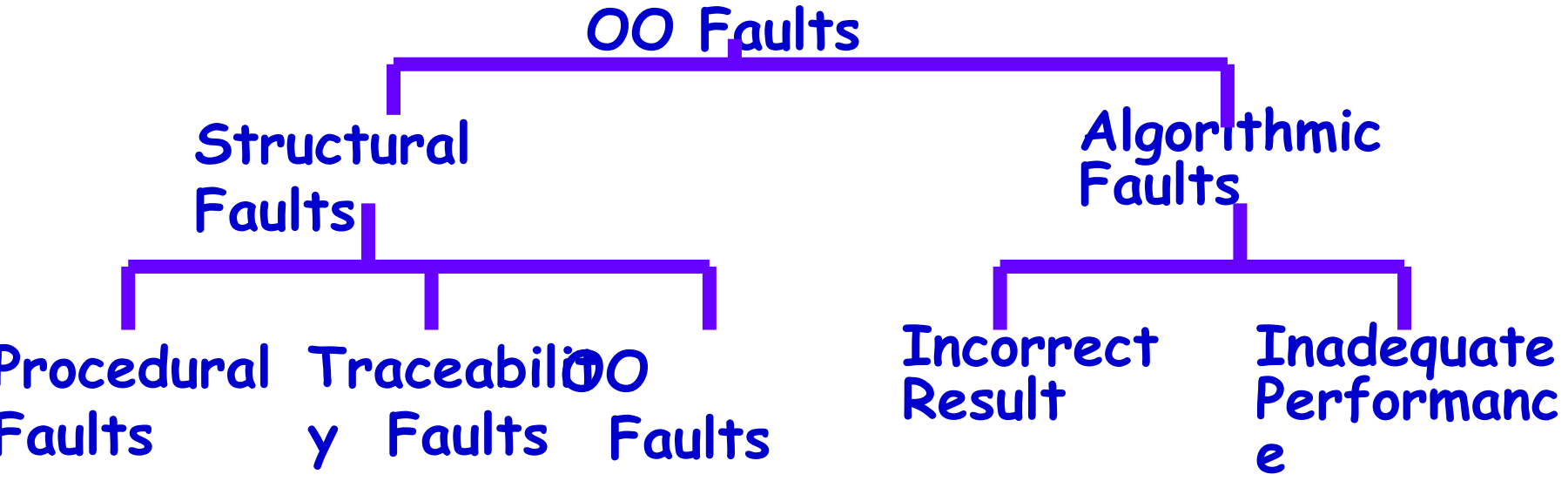
Pesticide Effect

- Assume we use 4 fault detection techniques and 1000 bugs:
  - Each detects only 70% bugs
  - How many bugs would remain
  - $1000*(0.3)^4=81$ bugs

Fault Model

- Types of faults possible in a program.
- Some types can be ruled out
  - Concurrency related-problems in a sequential program

Fault Model of an OO Program



Hardware Fault-Model

- Simple:
  - Stuck-at 0
  - Stuck-at 1
  - Open circuit
  - Short circuit
- Simple ways to test the presence of each
- Hardware testing is fault-based testing

Software Testing

- Each test case typically tries to establish correct working of some functionality
  - Executes (covers) some program elements
  - For restricted types of faults, fault-based testing exists.

Test Cases and Test Suites

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

Test Cases and Test Suites

- A 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.

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.

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 three main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach
  - Grey-box testing

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.

White-Box Testing

- There exist several popular white-box testing methodologies:
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - MC/DC coverage
  - Mutation testing
  - Data flow-based testing

Black-box Testing

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

- What does a ≡ b mod n mean?
- For a positive integer n, two integers a and b are said to be **congruent modulo n (or a is congruent to b modulo n), if a and b have the same remainder when divided by n (or equivalently if a – b is divisible by n)**.
- Congruence modulo n divides the set Z of all integers into n subsets called residue classes (equivalence classes).
- It can be expressed as a ≡ b mod n, n is called the modulus.

Equivalence class partitioning

- For example, if n = 2, then the two residue (equivalence) classes are the even integers and the odd integers.
- So, C1={0,2,4,6,8,10,…}
- C2={1,3,5,7,9,11,…}
- If, n = 3, then the residue (equivalence) classes are as follows:
- C1={0,3,6,9, 12,…}
- C2={1,4,7,10,13,…}
- C3={2,5,8,11,14,…}

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

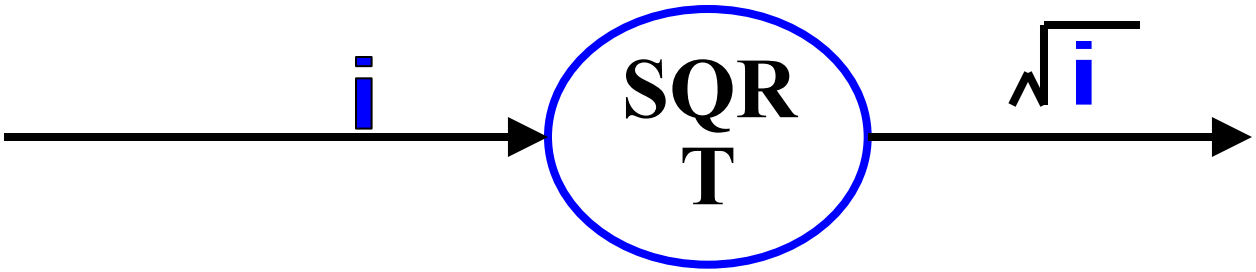
- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - one valid and two invalid equivalence classes are defined.

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

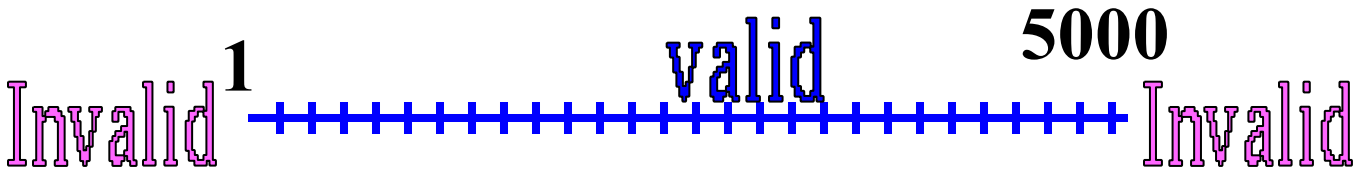


Example (cont.)

- 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}.



Example

- A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Solution

- 1. First we partition the domain of input as valid input values and invalid values, getting the following classes:
- I1 = {<A, B, C> : $1 \leq A \leq 50$}
- I2 = {<A, B, C> : $1 \leq B \leq 50$}
- I3 = {<A, B, C> : $1 \leq C \leq 50$}
- I4 = {<A, B, C> : $A < 1$}

Solution

- I5 = {<A, B, C> : A > 50}
- I6 = {<A, B, C> : B < 1}
- I7 = {<A, B, C> : B > 50}
- I8 = {<A, B, C> : C < 1}
- I9 = {<A, B, C> : C > 50}

Solution

- Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class.

Solution

- The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

Solution

- 2. We can derive another set of equivalence classes based on some possibilities for three integers, A, B, and C. These are given below:
- I1 = {<A, B, C> : A > B, A > C}
- I2 = {<A, B, C> : B > A, B > C}
- I3 = {<A, B, C> : C > A, C > B}

Solution

- I4 = {<A, B, C> : A = B, A ≠ C}
- I5 = {<A, B, C> : B = C, A ≠ B}
- I6 = {<A, B, C> : A = C, C ≠ B}
- I7 = {<A, B, C> : A = B = C}

Solution

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Example

- A program determines the next date in the calendar. Its input is entered in the form of with the following range:
- $1 \leq mm \leq 12$
- $1 \leq dd \leq 31$
- $1900 \leq yyyy \leq 2025$

Example

- Its output would be the next date or an error message ‘invalid date.’ Design test cases using equivalence class partitioning method.

Solution

- First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:
- $I1 = \{ \langle m, d, y \rangle : 1 \leq m \leq 12 \}$
- $I2 = \{ \langle m, d, y \rangle : 1 \leq d \leq 31 \}$
- $I3 = \{ \langle m, d, y \rangle : 1900 \leq y \leq 2025 \}$
- $I4 = \{ \langle m, d, y \rangle : m < 1 \}$

Solution

- I5 = {<m, d, y> : m > 12}
- I6 = {<m, d, y> : d < 1}
- I7 = {<m, d, y> : d > 31}
- I8 = {<m, d, y> : y < 1900}
- I9 = {<m, d, y> : y > 2025}

Solution

- The test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. The test cases are shown below:

Solution

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

Example

- A program takes an angle as input within the range [0, 360] and determines in which quadrant the angle lies. Design test cases using equivalence class partitioning method.

Solution

- 1. First we partition the domain of input as valid and invalid values, getting the follow
- I1 = {<Angle> : 0 ≤ Angle ≤ 360}
- I2 = {<Angle> : Angle < 0}
- I3 = {<Angle> : Angle > 360}

Solution

- The test cases designed from these classes are shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	I_1
2	–1	Invalid input	I_2
3	361	Invalid input	I_3

Solution

- 2. The classes can also be prepared based on the output criteria as shown below:
- O1 = {<Angle>: First Quadrant, if 0 ≤ Angle ≤ 90}
- O2 = {<Angle>: Second Quadrant, if 91 ≤ Angle ≤ 180}
- O3 = {<Angle>: Third Quadrant, if 181 ≤ Angle ≤ 270}

Solution

- O4 = {<Angle>: Fourth Quadrant, if 271 ≤ Angle ≤ 360}
- O5 = {<Angle>: Invalid Angle};
- However, O5 is not sufficient to cover all invalid conditions this way. Therefore, it must be further divided into equivalence classes as shown in next slide:

Solution

- O51 = {<Angle>: Invalid Angle, if Angle < 0}
- O52 = {<Angle>: Invalid Angle, if Angle > 360}

Solution

- Now the test cases can be designed from the above derived classes as shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	O_1
2	135	II Quadrant	O_2
3	250	III Quadrant	O_3
4	320	IV Quadrant	O_4
5	370	Invalid angle	O_{51}
6	–1	Invalid angle	O_{52}

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: {0,1,2,4999,5000,5001}.

- TS1 = {-5, 500, 6000} (EP)
- TS2 = {0,1,2,4999,5000,5001} (BVA)
- TS = TS1 U TS2
- ={-5,0,1,2,500,4999,5000,5001, 6000}

More Examples on Testing

Ex:-1

- Q. Check if 2 Straight Lines Intersect and Print Their Point of Intersection

  Ans: $y = mx + c$

Straight lines are given in the form of (m_1, c_1) and (m_2, c_2)

Step:1

Identify the Equivalent class

- ✓ Case 1: The lines are parallel i.e. $m_1 = m_2$

  So points are (1, 2) and (1, 5)

- ✓ Case 2: Coincident lines i.e. $m_1 = m_2$ and $c_1 = c_2$

  And points are (2, 3) and (2, 3)

Ex:-1 Contd.

$m_1 \neq m_2$

- ✓ Case 3: Lines intersecting at one point i.e.

The points may be (2, 5) and (3, 6).

So there are 3 equivalent classes.

There are no boundary values here.

So, Test Suite = {(1, 2),(1, 5),(2, 3),(2, 3),(2, 5),(3, 6)}

Ex:-2

- The Program Solves quadratic equations of the form

$$ax^2 + bx + c$$

It will accept 3 floating point values as input and it will gives the roots e.g. The input may be (7.7, 3.3 and 4.5).

Ans: Equivalent Classes

I. $b^2 = 4ac$ inputs are: a= 2.0, b=4.0 and c=2.0

II. $b^2 > 4ac$ inputs are a=2.0, b= 5.0 and c=2.0

III. $b^2 < 4ac$ inputs are a=2.0, b= 3.0 and c=2.0

IV. Invalid Equation inputs are a=0, b= 0 and c=10.0

So, Test Suite = {(2.0,4.0,2.0),(2.0,5.0,2.0),(2.0,3.0,2.0),(0.0,0.0,10.0)}

Ex:-3

- Solves linear equations in upto 10 independent variables

  e.g. 5x+6y+z=5

  10x+2y+5z=20

  ……

  ……

Example

- Equivalent Classes

I. Valid

   a. Many Solution (# var < #eqns)
   b. No Solution (# var > #eqns)
   c. Unique Solution (# var = #eqns)

II. Invalid

   a. Too many Variables (# var > 10)
   b. Invalid Equation (# var = 0)

Ex:-3

- Program Finds points of intersection of 2 circles
- Equivalent Classes

I. $r_1+r_2 <$ distance between(x_1, y_1) and (x_2, y_2) i.e. not intersecting

II. $r_1+r_2=$ distance i.e. touching at 1 point

III. $r_1+r_2>$ distance i.e. intersecting at 2 points

IV. Distance=0 and $r_1= r_2$ i.e. overlapping

V. Distance=0 and $r_1 \neq r_2$

VI. Invalid circles

Example: Query Book Option in LIS

Example: Testing the option Query Book using a Keyword (e.g. Author name or title).

The equivalent classes are

- Not present in catalogue (SE, not present)
- Present in catalogue (SE, present, 15 issued, not available)
- Present in catalogue (SE, present, 10 issued, 5 available)

Black Box testing

- Black-box testing attempts to find errors in the following categories:
- 1.To test the modules independently .
- 2. To test the functional validity of the software so that incorrect or missing functions can be recognized .
- 3. To look for interface errors.

Black Box testing

- 4. To test the system behavior and check its performance □
- 5. To test the maximum load or stress on the system.
- 6. To test the software such that the user/customer accepts the system within defined acceptable limits.

BOUNDARY VALUE ANALYSIS (BVA)

- BVA offers several methods to design test cases. Following are the few methods used:
- **1.** BOUNDARY VALUE CHECKING (BVC)
- **2.** ROBUSTNESS TESTING METHOD
- **3.** WORST-CASE TESTING METHOD
- **4.** ROBUST WORST-CASE TESTING METHOD

BOUNDARY VALUE CHECKING (BVC)

- In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

- The variable at its extreme value can be selected at:

BOUNDARY VALUE CHECKING (BVC)

- (a) Minimum value (Min)
- (b) Value just above the minimum value (Min+)
- (c) Maximum value (Max)
- (d) Value just below the maximum value (Max−)

BOUNDARY VALUE CHECKING (BVC)

- Let us take the example of two variables, A and B.
- If we consider all the above combinations with nominal values, then following test cases (see Fig. 1) can be designed:
- 1. Anom, Bmin
  2. Anom, Bmin+
  3. Anom, Bmax
  4. Anom, Bmax–
  5. Amin, Bnom
  6. Amin+, Bnom

BOUNDARY VALUE CHECKING (BVC)

- 7. Amax, Bnom
- 8. Amax–, Bnom
- 9. Anom, Bnom

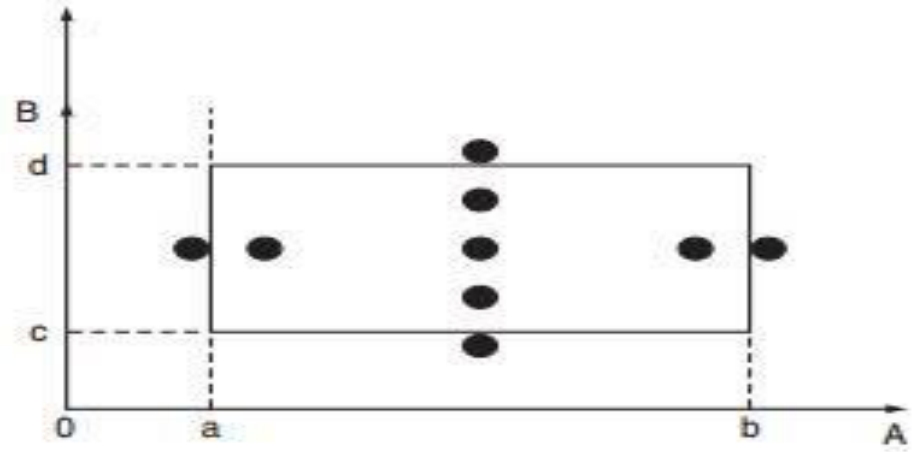


Fig: Boundary Value Checking

BOUNDARY VALUE CHECKING (BVC)

- It can be generalized that for n variables in a module, 4n + 1 test cases can be designed with boundary value checking method.

ROBUSTNESS TESTING METHOD

- The idea of BVC can be extended such that boundary values are exceeded as: □
- 1. A value just greater than the Maximum value (Max+)
- 2. A value just less than Minimum value (Min−)

ROBUSTNESS TESTING METHOD

- When test cases are designed considering the above points in addition to BVC, it is called robustness testing.
- Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:
- 10. Amax+, Bnom 11. Amin–, Bnom

ROBUSTNESS TESTING METHOD

- 12. Anom, Bmax+ 13. Anom, Bmin–

- It can be generalized that for n input variables in a module, 6n + 1 test cases can be designed with robustness testing.

WORST-CASE TESTING METHOD

- We can again extend the concept of BVC by assuming more than one variable on the boundary.
- It is called worst-case testing method.
- Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

WORST-CASE TESTING METHOD

- 10. Amin, Bmin
  11. Amin+, Bmin
  12. Amin, Bmin+
  13. Amin+, Bmin+
  14. Amax, Bmin
  15. Amax–, Bmin
  16. Amax, Bmin+
  17. Amax–, Bmin+
  18. Amin, Bmax
  19. Amin+, Bmax
  20. Amin, Bmax–
  21. Amin+,Bmax–
  22. Amax, Bmax
  23. Amax–, Bmax
  24. Amax, Bmax–
  25. Amax–,Bmax–

WORST-CASE TESTING METHOD

- It can be generalized that for n input variables in a module, 5n test cases can be designed with worst-case testing.

ROBUST WORST-CASE TESTING METHOD

- In the previous method, the extreme values of a variable considered are of BVC only.
- The worst case can be further extended if we consider robustness also, that is, in worst case testing if we consider the extreme values of the variables as in robustness testing method covered in Robustness Testing

ROBUST WORST-CASE TESTING METHOD

- Again take the example of two variables, A and B. We can add the following test cases to the list of 25 test cases designed in previous section.
- 26. Amin-, Bmin-
- 27. Amin-, Bmin
- 28. Amin, Bmin-
- 29. Amin-, Bmin+
- 30. Amin+, Bmin-
- 31. Amin-, Bmax

Solution

- 32. Amax, Bmin-
- 33. Amin-, Bmax-
- 34. Amax-, Bmin-
- 35. Amax+,Bmax+
- 36. Amax+, Bmin
- 37. Amin, Bmin+
- 38. Amax+, Bmin+
- 39. Amax+,Bmax+
- 40. Amax+,Bmax
- 41. Amax, Bmax+
- 42. Amax+,Bmax-
- 43. Amax-, Bmax+
- 44. Amax+,Bnom
- 45. Anom, Bmax+

Solution

- 46. Amin-,Bnom
- 47. Anom, Bmin-
- 48. Amax+,Bmin-
- 49. Amin-, Bmax+

STATE TABLE – BASED TESTING

- Tables are useful tools for representing and documenting many types of information relating to test case design.
- Theses are beneficial for applications which can be described using state transition diagrams and state tables.

Basic terms related to State Table

1. Finite State Machine (FSM)

- An FSM is a behavioral model whose outcome depends upon both the previous and current inputs.
- This model can be prepared for software structure or software behavior.
- It can be used as a tool for functional testing.

2. State Transition Diagrams or State Graph

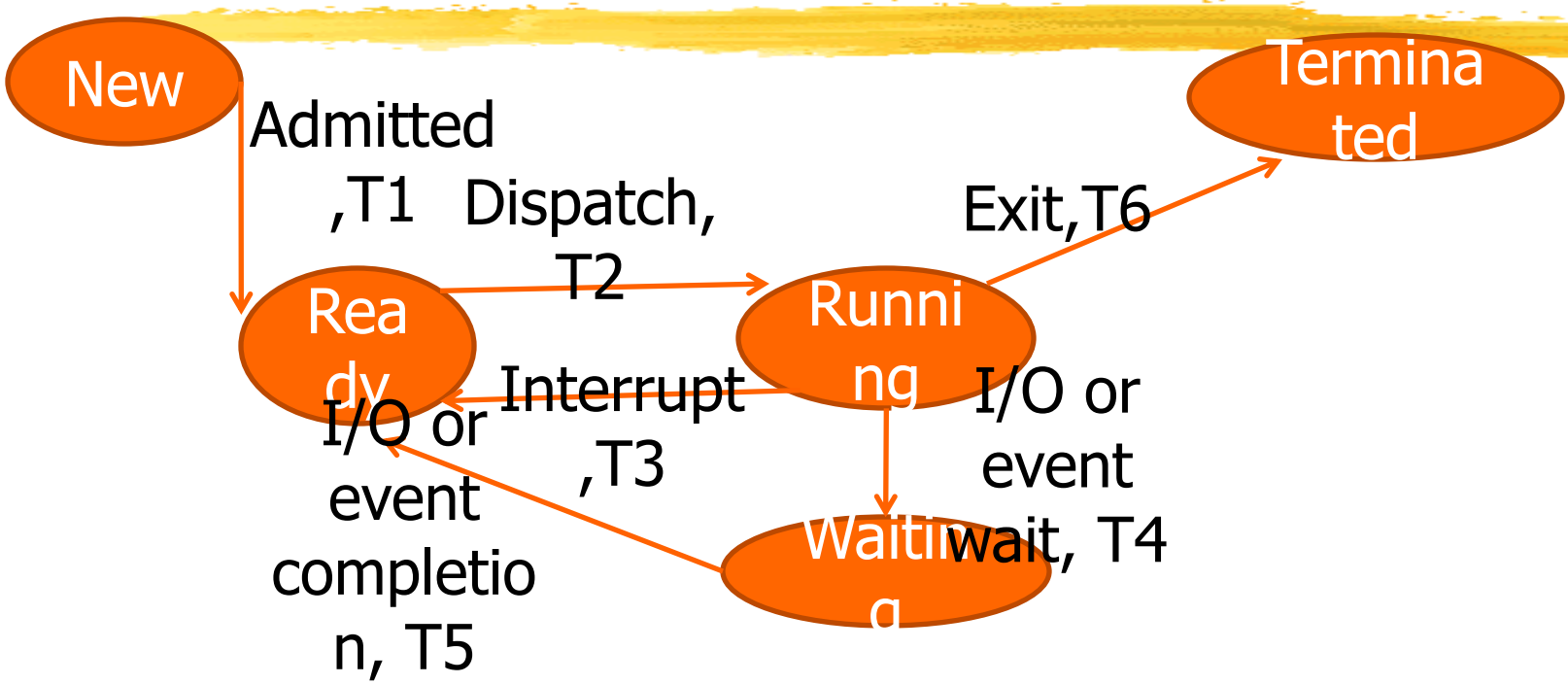
- A system or its components may have a number of states depending on its *input and time.*
- States are represented by *nodes.*
- *With the help of nodes and transition links between nodes, a STD or SG can be prepared.*
- A state graph is the pictorial representation of an FSM.
- Its purpose is to depict the states that a system or its components can assume.
- It shows the events or circumstances that cause or result from a change from one state to another.

- Whatever is being modeled is subjected to inputs.
- As a result of these inputs, when one state is changed to another is called a *transition.*
- Transitions are represented by links that join the nodes.

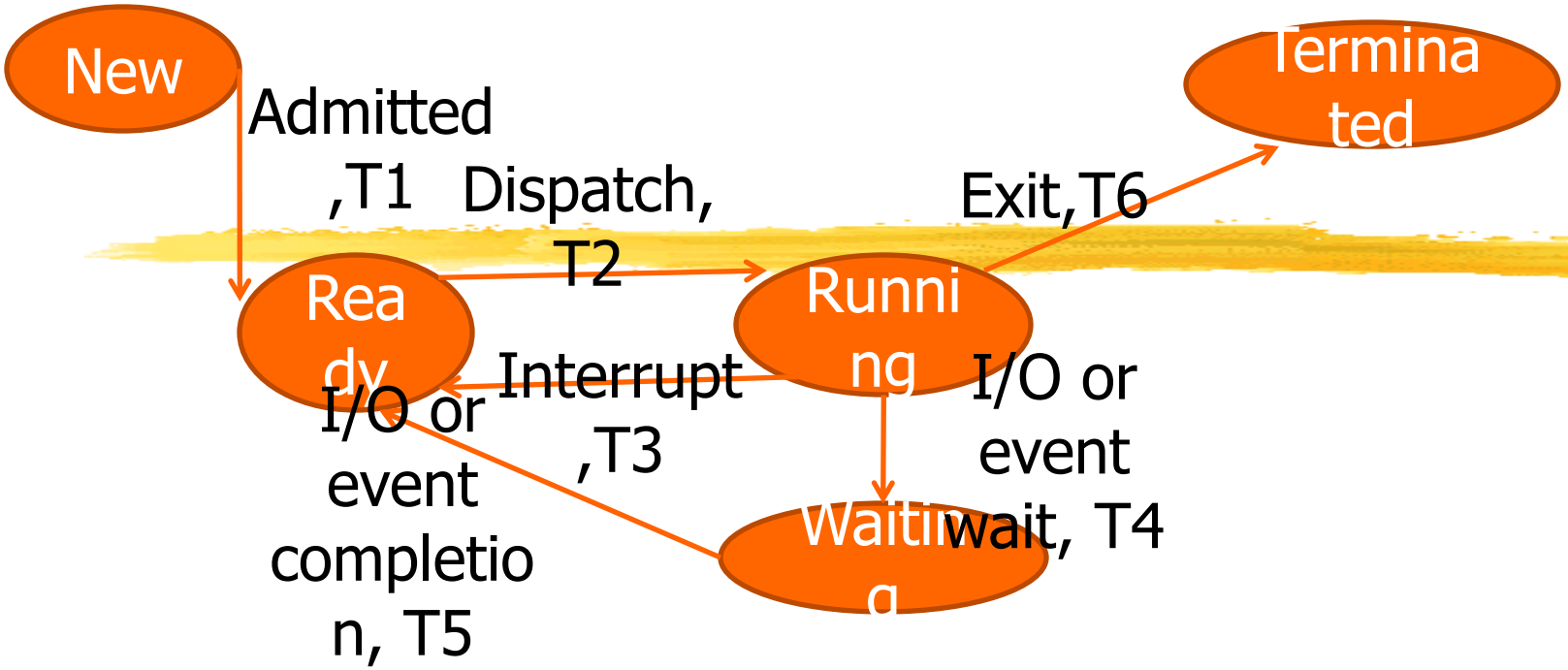
For example, a task in an O. S. can have its following states:

i. **New State :** When a task is newly created
ii. **Ready :** When the task is waiting in the ready queue for its turn.
iii. **Running :** When Instructions of the task are being executed by CPU.
iv. **Waiting :** When the task is waiting for an I/O event or reception of a signal
v. **Terminated :** The task has finished execution.

State Graph



i. New State : When a task is newly created
ii. Ready : When the task is waiting in the ready queue for its turn.
iii. Running : When Instructions of the task are being executed by CPU.
iv. Waiting : When the task is waiting for an I/O event or reception of a signal
v. Terminated : The task has finished execution.



Each arrow link provides two types of Information :

1. **Transition events like admitted, dispatch, interrupt, etc.**
2. **The resulting output from a state like T1, T2, T3 etc.**

T0=Task is in new state and waiting for admission to ready queue.
T1= A new task admitted to ready queue
T2= A ready task has started running
T3= Running task has been interrupted
T4= Running task is waiting for I/O or event
T5= Wait period of waiting task is over
T6= Task has completed execution

3. State Table

State/input Event	Admit	Dispatch	Interrupt	I/O or event Wait	I/O or event Wait Over	Exit
New	**Ready/T1**	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready / T1	**Running /T2**	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running /T2	Running /T2	**Ready / T3**	**Waiting/T4**	Running/T2	**Terminated/T6**
Waiting	Waiting/T4	Waiting/T4	Waiting/T4	Waiting/T4	**Ready /T5**	Waiting/T4

- Each rows of the table corresponds to a state.
- Each column corresponds to an input condition
- The box at the intersection of a row and a column specifies the next state (transition) and the outputs, if any.

4. State Table-Based Testing

- After reviewing the basics, we can start functional testing with state tables.

Steps:

1. Identify the states

The number of states in a state graph is the number of states we choose to recognize or model.

Find the number states as follows :

- Identify all the component factors of the state.
- Identify all the allowable values for each factor
- The number of states is the product of the number of allowable values of all the factors.

2. Prepare state transition diagram after understanding transitions between states

- After having all the states, identify the inputs on each state and transitions between states and prepare the state graph.
- Every input state combination must have a specified transition.

3. Convert the state graph into the state table as discussed earlier

4. Analyze the state table for its completeness.

5. Create the corresponding test cases from the state table

Test case ID	Test Source	Input		Expected results	
		Current State	Event	Output	Next state
TC1	Cell 1	New	Admit	T1	Ready
TC2	Cell 2	New	Dispatch	T0	New
TC3	Cell 3	New	Interrupt	T0	New
TC4	Cell 4	New	I/O wait	T0	New
TC5	Cell 5	New	I/O wait over	T0	New
TC6	Cell 6	New	Exit	T0	New
TC7	Cell 7	Ready	Admit	T1	Ready
TC8	Cell 8	Ready	Dispatch	T2	Running
TC9	Cell 9	Ready	Interrupt	T1	Ready
TC10	Cell 10	Ready	I/O wait	T1	Ready
TC11	Cell 11	Ready	I/O wait over	T1	Ready
TC12	Cell 12	Ready	Exit	T1	Ready
TC13	Cell 13	Running	Admit	T2	Running
TC14	Cell 14	Running	Dispatch	T2	Running
TC15	Cell 15	Running	Interrupt	T3	Ready
TC16	Cell 16	Running	I/O wait	T4	Waiting
TC17	Cell 17	Running	I/O wait over	T2	Running
TC18	Cell 18	Running	Exit	T6	Terminated
TC19	Cell 19	Waiting	Admit	T4	Waiting
TC20	Cell 20	Waiting	Dispatch	T4	Waiting
TC21	Cell 21	Waiting	Interrupt	T4	Waiting
TC22	Cell 22	Waiting	I/O wait	T4	Waiting
TC23	Cell 23	Waiting	I/O wait over	T5	Ready
TC24	Cell 24	Waiting	Exit	T4	Waiting

- Test cases are produced in a tabular form known as the test case.
- **Test cases ID :** a unique identifier for each test case
- **Test Source :** a trace back to the corresponding cell in the state table.
- **Current state :** the initial condition to run the test
- **Event :** the input triggered by the user
- **Output :** the current value returned
- **Next State :** the new state achieved

Decision Table – Based Testing

- Boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions.
- There may be some critical behaviour to be tested when some combinations of input conditions are considered.
- Decision table is another useful method to represent the information in a tabular method.
- Decision table has the specialty to consider complex combinations of input conditions and resulting actions.
- Decision tables obtain their power from logical expressions.
- Each operand or variable in a logical expression takes on the value, TRUE or FALSE

Formation of Decision Table

		Rule 1	Rule 2	Rule 3	Rule 4	…
Condition Stub	C1 C2 C3	True False True	True True True	False False True	I True I	
Action Stub	A1 A2 A3	X	X	X	X	

Condition stub It is a list a list of input conditions for which the complex combination is made.

Action stub It is a list of resulting action which will be performed if a combination of input condition is satisfied.

Condition entry

- It is a specific entry in the table corresponding to input conditions mentioned in the condition stub.
- When the condition entry takes only two values – TRUE or FALSE then it is called ***Limited Entry Decision Table.***
- When the condition entry takes several values , then it is called ***Extended Entry Decision Table.***

Action entry It is the entry in the table for the resulting action to be performed

- List all actions that can be associated with a specific procedure.
- List all conditions during execution of the procedure.
- Associate specific sets of conditions with specific actions, eliminating impossible combinations and conditions; alternatively, develop every possible permutation of conditions.
- Define rules by indicating what action occurs for a set of conditions.

Test Case Design using Decision Table

- Interpret condition stubs as the inputs for the test case.
- Interpret action stubs as the expected output for the test case.
- Rule, which is the combination of input conditions, becomes the test case itself.
- If there are *k* rules over *n* binary conditions, there are at least *k* test cases and at the most 2^n test cases.

Example

A programme calculates the total salary of an emplyee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design test cases using decision table testing.

Solution

Entry

		Rule 1	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48 C2: Holidays or Sundays	I T	F F	T F
Action Stub	A1: Normal Salary A2: 1.25 of salary A3: 2.00 of salary	 X	X 	 X

Decision Table

Test case ID	Working hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

Expanding the Immaterial Cases in Decision Table

- These conditions means that the value of a particular condition in the specific rule does not make a difference whether it is TRUE or FALSE.
- Sometimes expanding the decision table to spell out don't-care conditions can reveal hidden problems.

Example

Entry (Decision table)

		Rule 1	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48 C2: Holidays or Sundays	I T	F F	T F
Action Stub	A1: Normal Salary A2: 1.25 of salary A3: 2.00 of salary	 X	X 	 X

The immaterial test case in rule 1 of the above table can be expanded by taking both T and F values of C1.

Entry (Expanded decision table)

		Rule 1-1	Rule 1-2	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48 C2: Holidays or Sundays	F T	T T	F F	T F
Action Stub	A1: Normal Salary A2: 1.25 of salary A3: 2.00 of salary	X	X	X	X

Entry (Expanded decision table)

Test case ID	Working hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary
4	30	Sunday	2.00 of salary

Orthogonal Array Testing

- Orthogonal Arrays are two dimensional arrays of numbers that have the attractive feature that by selecting any two columns in the array, an even distribution of all pairwise combinations of values in the array can be achieved.

Example

- In the following table , after selecting first two columns, we get four ordered pairs namely (0,0), (1,1), (0,1) and (1,0).
- These pairs form all the possible ordered pairs of two-element set and each ordered pair appears exactly once.

Example

0	0	0
1	1	0
0	1	1
1	0	1

We obtain same values when selecting second and third or first and third column combination. An array exhibiting this feature is known as orthogonal array.

Orthogonal Array Testing

- Orthogonal arrays can be used in software testing for pairwise interactions.
- It provides uniformly distributed coverage for all variable pairwise combinations.
- It is commonly used for integration testing like object oriented systems, where multiple subclasses can be substituted as the server for a client.

Orthogonal Array Testing

- It is black-box testing technique.
- OATS is used when the input to the system to be tested are low but if exhaustive testing is used then it is not possible to test completely every input of the system.
- 100% OATS implies 100% pairwise testing.
- OATS can be used for testing combinations of configurable options like a webpage that allows the other user to select:

Orthogonal Array Testing

- Font style;
- Font color;
- Back ground Color;
- Page layout;
- Etc.

Steps to use OATS

- Step 1: Identify the independent variables that are to be used for interaction. These will be mapped as “factor” (f) of array.
- Step 2: Identify the maximum number of values, which each variable will take. There will be mapped as “levels” (p) of the array.
- Search for an orthogonal array that has all factors from step 1 and all levels from step 2.

Steps to use OATS

- Step 4: Map all the factors and levels with your requirements.
- Translate them into suitable test cases.
- Look out for any special test cases.
- If we have 3 variables (parameters) that we have 3 value then the possible number of test cases using conventional technique is 3*3*3=27 but if OATS is used, then number of test cases will be 9.

Example 1

- Consider a scenario in which we need to derive test cases for a web page of a research paper that has four different sections:
- (a) Abstract
- (b) Related work
- (c) Proposed work
- (d) Conclusion

Example 1

- The four section can be individually shown or hidden to the user or show message. Thus it is required to design the test condition to test interaction between different sections

Solution

- Step 1: Number of independent variables or factors = 4.
- Step 2: Value that each independent variable can take = 3 values (shown , hidden or error message).
- Step 3: Orthogonal array would be 3^2 .
- Step 4: The appropriate orthogonal array for 4 factors and 3 levels is shown in the table.

Solution

Experiment No.	Factor A	Factor B	Factor C	Factor D
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Solution

- Step 5: Now, map the table with our requirements.
- 1 will represent “Shown” value.
- 2 will represent “Hidden” value.
- 3 will represent “Error Message” value.
- Factor A will represent “Abstract” section.
- Factor B will represent “Related Work” section.
- Factor C will represent “Proposed Work” section.

Solution

- Factor A will represent “Conclusion” section.
- Experiment will represent “Test Case #”.
- Step 6: After mapping, the table will look like:

Solution

Test Case	Abstract	Related Work	Proposed work	Conclusion
Test Case 1	Shown	Shown	Shown	Shown
Test Case 2	Shown	Hidden	Hidden	Hidden
Test Case 3	Shown	Error Message	Error Message	Error Message
Test Case 4	Hidden	Shown	Hidden	Error Message
Test Case 5	Hidden	Hidden	Error Message	Shown
Test Case 6	Hidden	Error Message	Shown	Hidden
Test Case 7	Error Message	Shown	Error Message	Hidden
Test Case 8	Error Message	Hidden	Shown	Error Message
Test Case 9	Error Message	Error Message	Hidden	Shown

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.

Example

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

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.

Program Analysis Tools

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

Program Analysis Tools

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

Static Analysis Tools

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

Static Analysis Tools

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

Static Analysis Tools

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

Dynamic Analysis Tools

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

Testing

- The aim of testing is to identify all defects in a software product.
- However, in practice even after thorough testing:
  - one cannot guarantee that the software is error-free.

Testing

- The input data domain of most software products is very large:
  - it is not practical to test the software exhaustively with each input data value.

Testing

- Testing does however expose many errors:
  - testing provides a practical way of reducing defects in a system
  - increases the users' confidence in a developed system.

Testing

- Testing is an important development phase:
  - requires the maximum effort among all development phases.
- In a typical development organization:
  - maximum number of software engineers can be found to be engaged in testing activities.

Testing

- Many engineers have the wrong impression:
  - testing is a secondary activity
  - it is intellectually not as stimulating as the other development activities, etc.

Testing

- Testing a software product is in fact:
  - as much challenging as initial development activities such as specification, design, and coding.
- Also, testing involves a lot of creative thinking.

Testing

- Software products are tested at three levels:
  - Unit testing
  - Integration testing
  - System testing

Unit testing

- During unit testing, modules are tested in isolation:
  - If all modules were to be tested together:
    - it may not be easy to determine which module has the error.

Unit testing

- Unit testing reduces debugging effort several folds.
  - Programmers carry out unit testing immediately after they complete the coding of a module.

Integration testing

- After different modules of a system have been coded and unit tested:
  - modules are integrated in steps according to an integration plan
  - partially integrated system is tested at each integration step.

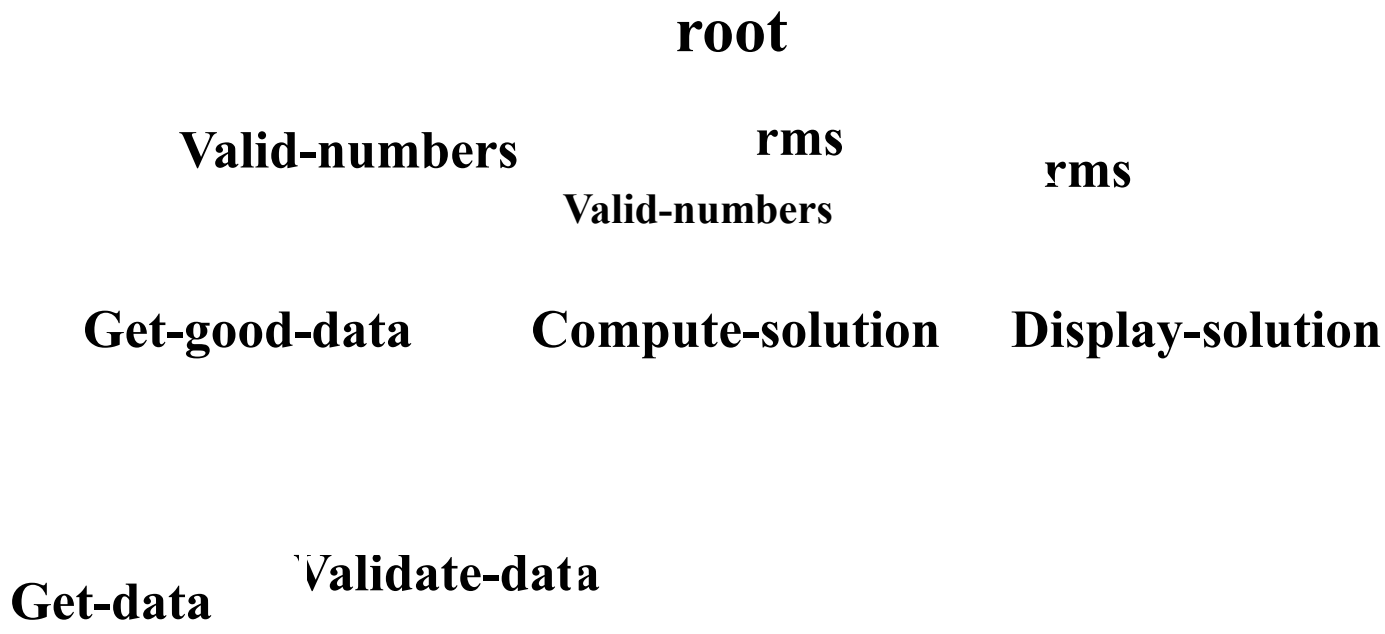
System Testing

- System testing involves:
  - validating a fully developed system against its requirements.

Integration Testing

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach

Example Structured Design



Big bang Integration Testing

- Big bang approach is the simplest integration testing approach:
  - all the modules are simply put together and tested.
  - this technique is used only for very small systems.

Big bang Integration Testing

- Main problems with this approach:
  - if an error is found:
    - it is very difficult to localize the error
    - the error may potentially belong to any of the modules being integrated.
  - debugging errors found during big bang integration testing are very expensive to fix.

Bottom-up Integration Testing

- Integrate and test the bottom level modules first.
- A disadvantage of bottom-up testing:
  - when the system is made up of a large number of small subsystems.
  - This extreme case corresponds to the big bang approach.

Top-down integration testing

- Top-down integration testing starts with the main routine:
  - and one or two subordinate routines in the system.
- After the top-level 'skeleton' has been tested:
  - immediate subordinate modules of the 'skeleton' are combined with it and tested.

Mixed integration testing

- Mixed (or sandwiched) integration testing:
  - uses both top-down and bottom-up testing approaches.
  - Most common approach

Integration Testing

- In top-down approach:
  - testing waits till all top-level modules are coded and unit tested.
- In bottom-up approach:
  - testing can start only after bottom level modules are ready.

System Testing

- There are three main kinds of system testing:
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing

Alpha Testing

- System testing is carried out by the test team within the developing organization.

Beta Testing

- System testing performed by a select group of friendly customers.

Acceptance Testing

- System testing performed by the customer himself:
  - to determine whether the system should be accepted or rejected.

Stress Testing

- Stress testing (aka endurance testing):
  - impose abnormal input to stress the capabilities of the software.
  - Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

How many errors are still remaining?

- Seed the code with some known errors:
  - artificial errors are introduced into the program.
  - Check how many of the seeded errors are detected during testing.

Error Seeding

- Let:
  - N be the total number of errors in the system
  - n of these errors be found by testing.
  - S be the total number of seeded errors,
  - s of the seeded errors be found during testing.

Error Seeding

- $n/N = s/S$
- $N = S\ n/s$
- remaining defects:

$$N - n = n\ ((S - s)/ s)$$

Example

- 100 errors were introduced.
- 90 of these errors were found during testing
- 50 other errors were also found.
- Remaining errors=
  50 (100-90)/90 = 6

Error Seeding

- The kind of seeded errors should match closely with existing errors:
  - However, it is difficult to predict the types of errors that exist.
- Categories of remaining errors:
  - can be estimated by analyzing historical data from similar projects.

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.