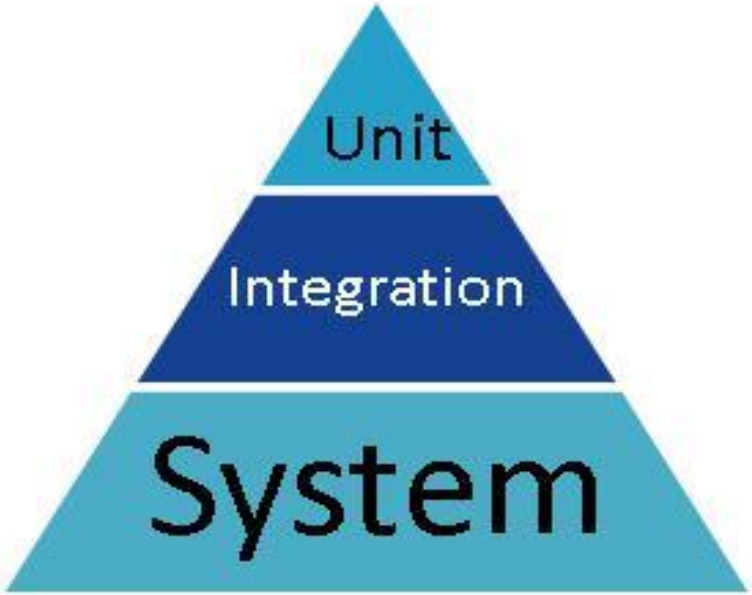
Software Testing

by
Dr. Ashima Singh
Associate Professor, CSED

TESTING

What is testing?

Testing is the process of evaluating a system or its component(s) with the intent to find that whether it satisfies the specified requirements or not. This activity results in the actual, expected and difference between their results. In simple words testing is executing a system in order to identify any gaps, errors or missing requirements in contrary to the actual desire or requirements.



According to ANSI/IEEE 1059 standard, Testing can be defined as "A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item".

Software Failure

- A requirement was omitted
- Requirements were not possible to implement
- Specification was incorrect
- Error in the high-level design (architecture)
- Error in the low-level design
- Error in the code

Terminology

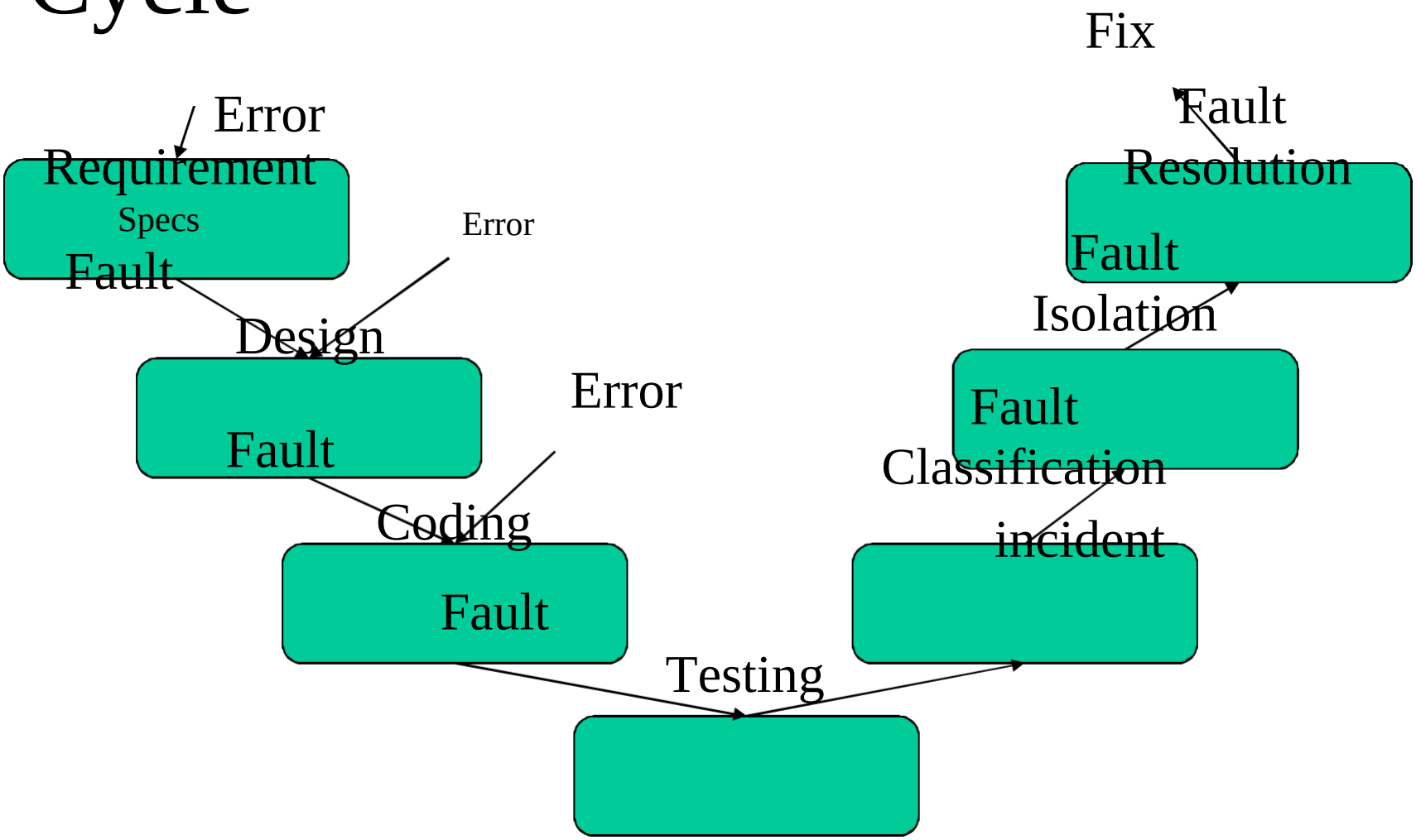
- Testability: The effort required to test a software to ensure that it performs its intended functions.
- *Failure:* misbehavior of the system

- *Fault:* An error in the system that eventually causes a failure, also know as a “bug”

- *Fault identification:* the process of discovering what fault caused a failure

- *Fault correction:* the process of ensuring that a failure does not happen again

A Testing Life Cycle



Terminology

- Error
  - Represents mistakes made by people
- Fault
  - Is result of error. May be categorized as
    - Fault of Commission – we enter something into representation that is incorrect
    - Fault of Omission – Designer can make error of omission, the resulting fault is that something is missing that should have been present in the representation.

- Incident
  - Behavior of fault. An incident is the symptom(s) associated with a failure that alerts user to the occurrence of a failure
- Test case
  - Associated with program behavior. It carries set of input and list of expected output

Who Does the Testing

- It depends on the process and the associated stakeholders of the project(s). In the IT industry, large companies have a team with responsibilities to evaluate the developed software in the context of the given requirements. Moreover, developers also conduct testing which is called Unit Testing. In most cases, following professionals are involved in testing of a system within their respective capacities:
- Software Tester
- Software Developer
- Project Lead/Manager
- End User

When to Start Testing

- An early start to testing reduces the cost, time to rework and error free software that is delivered to the client.
- However in Software Development Life Cycle (SDLC) testing can be started from the Requirements Gathering phase and lasts till the deployment of the software. It also depends on the development model that is being used.
- For example in Water fall model formal testing is conducted in the Testing phase, but in incremental model, testing is performed at the end of every increment/iteration and at the end the whole application is tested.
- Testing is done in different forms at every phase of SDLC like during Requirement gathering phase, the analysis and verifications of requirements are also considered testing.
- Reviewing the design in the design phase with intent to improve the design is also considered as testing.
- Testing performed by a developer on completion of the code is also categorized as Unit type of testing.

When to Stop Testing

- Unlike when to start testing it is difficult to determine when to stop testing, as testing is a never ending process and no one can say that any software is 100% tested. Following are the aspects which should be considered to stop the testing:
- Testing Deadlines.
- Completion of test case execution.
- Completion of Functional and code coverage to a certain point.
- Bug rate falls below a certain level and no high priority bugs are identified.
- Management decision.

Testing=Verification + Validation

These two terms are very confusing for people, who use them interchangeably. Let's discuss about them briefly.

Verification	Validation
Are you building it right?	Are you building the right thing?
Ensure that the software system meets all the functionality.	Ensure that functionalities meet the intended behavior.
Verification takes place first and includes the checking for documentation, code etc.	Validation occurs after verification and mainly involves the checking of the overall product.
Done by developers.	Done by Testers.
Have static activities as it includes the reviews, walkthroughs, and inspections to verify that software is correct or not.	Have dynamic activities as it includes executing the software against the requirements.
It is an objective process and no subjective decision should be needed to verify the Software.	It is a subjective process and involves subjective decisions on how well the Software works.

Contd

- Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests.

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers.

Alpha Tests are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

Comparison: Quality Assurance ,Quality Control and Testing

Quality Assurance	Quality Control	Testing
Activities which ensure the implementation of processes, procedures and standards in context to verification of developed software and intended requirements.	Activities which ensure the verification of developed software with respect to documented (or not in some cases) requirements.	Activities which ensure the identification of bugs/error/defects in the Software.
Focuses on processes and procedures rather then conducting actual testing on the system.	Focuses on actual testing by executing Software with intend to identify bug/defect through implementation of procedures and process.	Focuses on actual testing.
Process oriented activities.	Product oriented activities.	Product oriented activities.
Preventive activities	It is a corrective process.	It is a preventive process
It is a subset of Software Test Life Cycle (STLC)	QC can be considered as the subset of Quality Assurance.	Testing is the subset of Quality Control.

Difference between Testing and Debugging

Testing: It involves the identification of bug/error/defect in the software without correcting it. Normally professionals with a Quality Assurance background are involved in the identification of bugs. Testing is performed in the testing phase.

Debugging: It involves identifying, isolating and fixing the problems/bug. Developers who code the software conduct debugging upon encountering an error in the code. Debugging is the part of White box or Unit Testing. Debugging can be performed in the development phase while conducting Unit Testing or in phases while fixing the reported bugs**.**

Test Case

- Test case is a triplet [I, S, O] where
  - I is input data
  - S is state of system at which data will be input
  - O is the **expected** output
- Test suite is set of all test cases.
- Test cases are not randomly selected. Instead even they need to be designed.

Need for Designing Test Cases

- Almost every non-trivial system has an extremely large input data domain thereby making exhaustive testing impractical

- If randomly selected then test case may loose significance since it may expose an already detected error by some other test case

Design of Test Cases

- Number of test cases do not determine the effectiveness.
- To detect error in following code

  if(x>y) max = x; else max = x;
- {(x=3, y=2); (x=2, y=3)} will suffice.
- {(x=3, y=2); (x=4, y=3); (x=5, y = 1)} will falter.
- Each test case should detect different errors

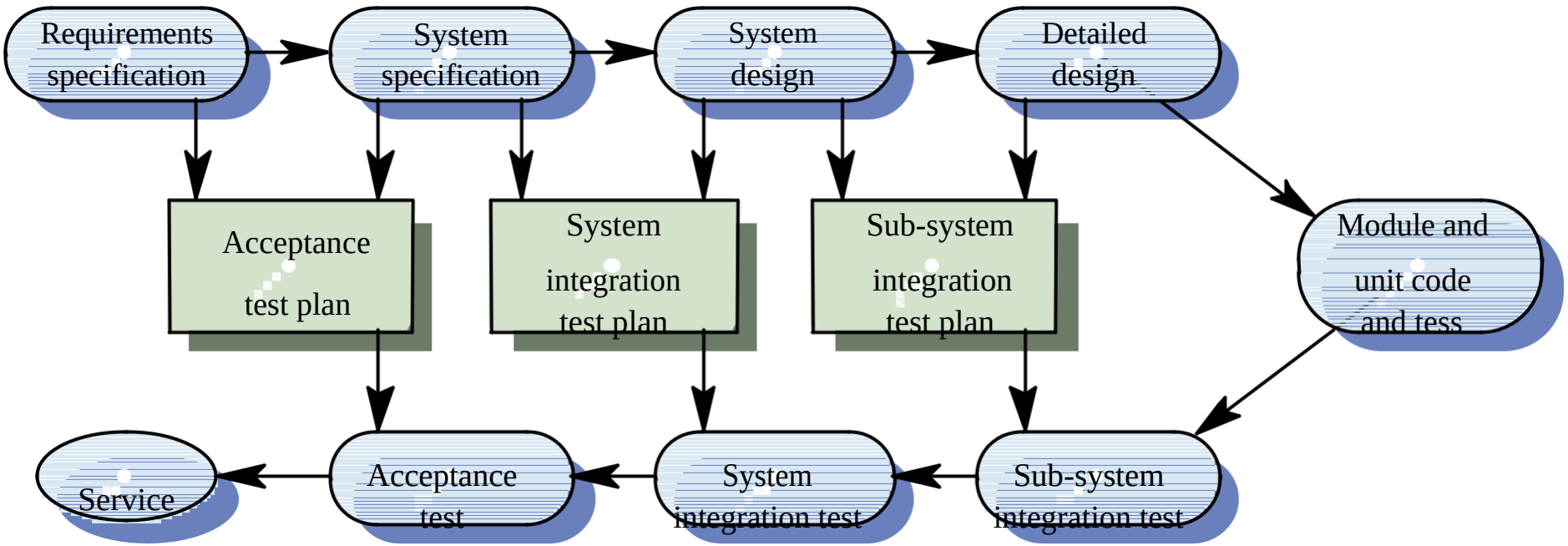
Test Case

Test Case ID:	<TC ID>	**Test Engineer:**	<Test Engineer>
Product Module:	Home Page	**Testing Date:**	29-08-2011
Product Version:		**Testing Cycle:**	1
Revision History:		**Status:**	
Purpose:	<Purpose>		
Assumptions	<Assumptions>		
Pre-Conditions:	<Pre-Condition>		
Steps to Reproduce:	<Steps to Reproduce>		
Expected Results:	<Expected Outcome>		
Actual Outcome:	<Actual Outoome>		
Post Conditions:	<Purpose>		

The Testing Process

- Is a whole life-cycle process.
- Testing (static or dynamic) must be applied at each stage in the software process.
- Has two principal complementary objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation.

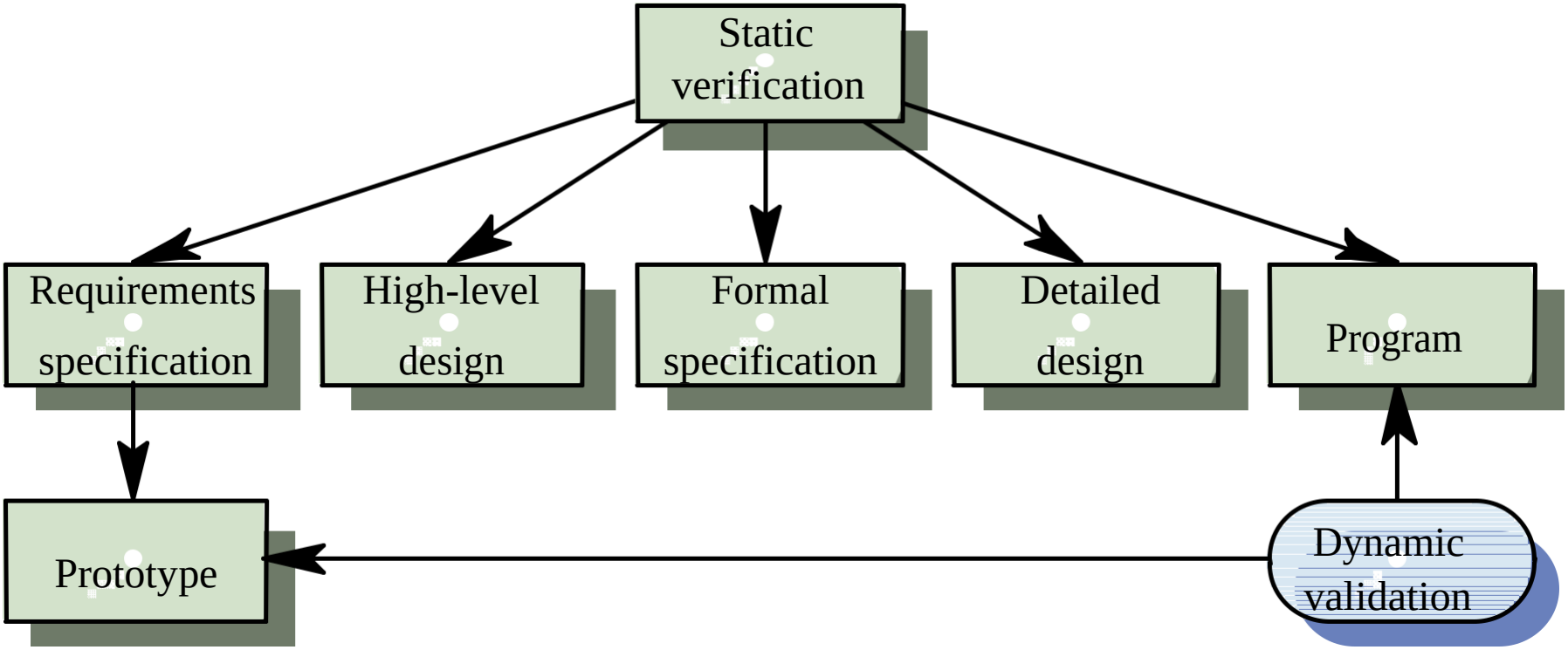
The V-model of development



Static and dynamic testing

- *Static testing*
  - Concerned with analysis of the static system representation (program & documentation) to discover problems
- *Dynamic testing*
  - Concerned with exercising and observing product behavior

Static and dynamic testing



Static testing

Static Testing

- Involves analysis of source text by humans or software
- Can be carried out on ANY documents produced as part of the software process
- Discovers errors early in the software process
- Usually more cost-effective than testing for defect detection at the unit and module level
- Allows defect detection to be combined with other quality checks

Static testing effectiveness

- More than 60% of program errors can be detected by informal program inspections
- More than 90% of program errors may be detectable using more formal program verification

Program inspections

- Formalized approach to document reviews
- Intended explicitly for defect detection, not correction
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialized variable) or non-compliance with standards

Inspection procedure

- System overview presented to inspection team
- Code and associated documents are distributed to inspection team in advance
- Inspection takes place and discovered errors are noted
- Modifications are made to repair discovered errors
- Re-inspection may or may not be required

Formal Inspection

- Formal Inspection meetings may have following process:

- Planning
- Overview Preparation
- Inspection Meeting
- Rework
- Follow-up.

Inspection teams

- Made up of at least 5 members
  - Author of the code being inspected
  - Reader who reads the code to the team
  - Inspector who finds errors, omissions and inconsistencies
  - Moderator who chairs the meeting and notes discovered errors
  - Scribe taking notes on the inspection process results

Static analysis checks

Fault class	Static analysis check
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

Dynamic Testing

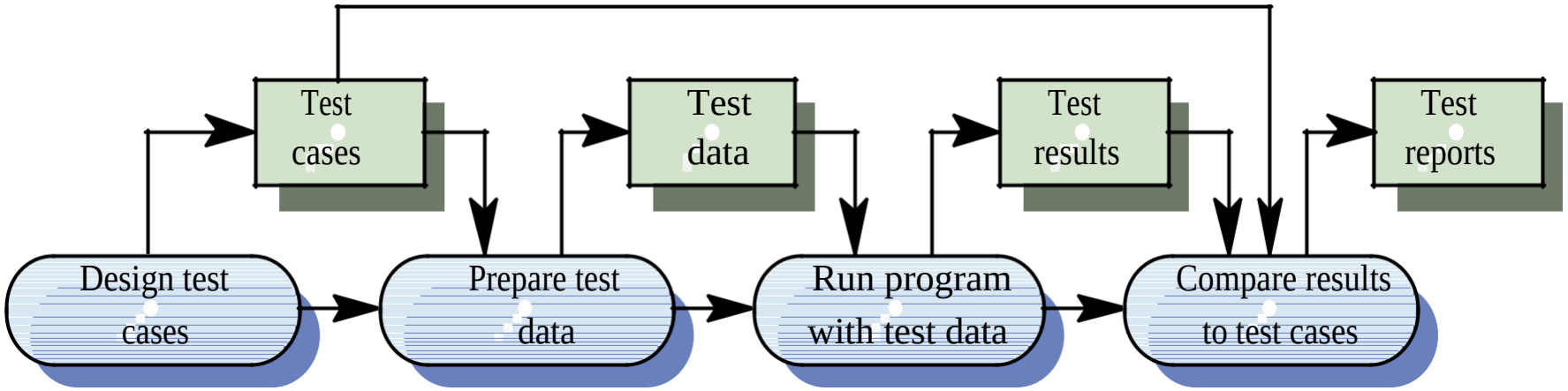
Dynamic Testing

- The objective of dynamic testing is to discover **defects in running systems**
- A successful defect test is a test which causes a program to behave in an anomalous way
- Only validation technique **for non-functional requirements**
- Should be used in conjunction with static testing

Testing at different levels and stages

- ***Unit testing:*** testing each module in isolation. Involves the creation of a testing environment for providing input/collecting output to the component.
- ***Integration testing:*** testing groups of components incrementally. Defining the order of integration is of prime importance.
- ***Function testing:*** functionalities of the system are tested using the whole system
- ***Performance testing:*** Testing the system to the limits of its operational requirements, and beyond.
- ***Acceptance testing:*** The client is invited to test the software for acceptance (alpha testing)
- ***Installation testing:*** After installation on-site, the software is tested again.
- ***Beta testing:*** the software is given to many customers for them to install and test with no test guidance at all.

The dynamic testing process



Classification of Test

- There are two levels of classification
  - One distinguishes at granularity level
    - Unit level
    - System level
    - Integration level
  - Other classification (mostly for unit level) is based on methodologies
    - Black box (Functional) Testing
    - White box (Structural) Testing

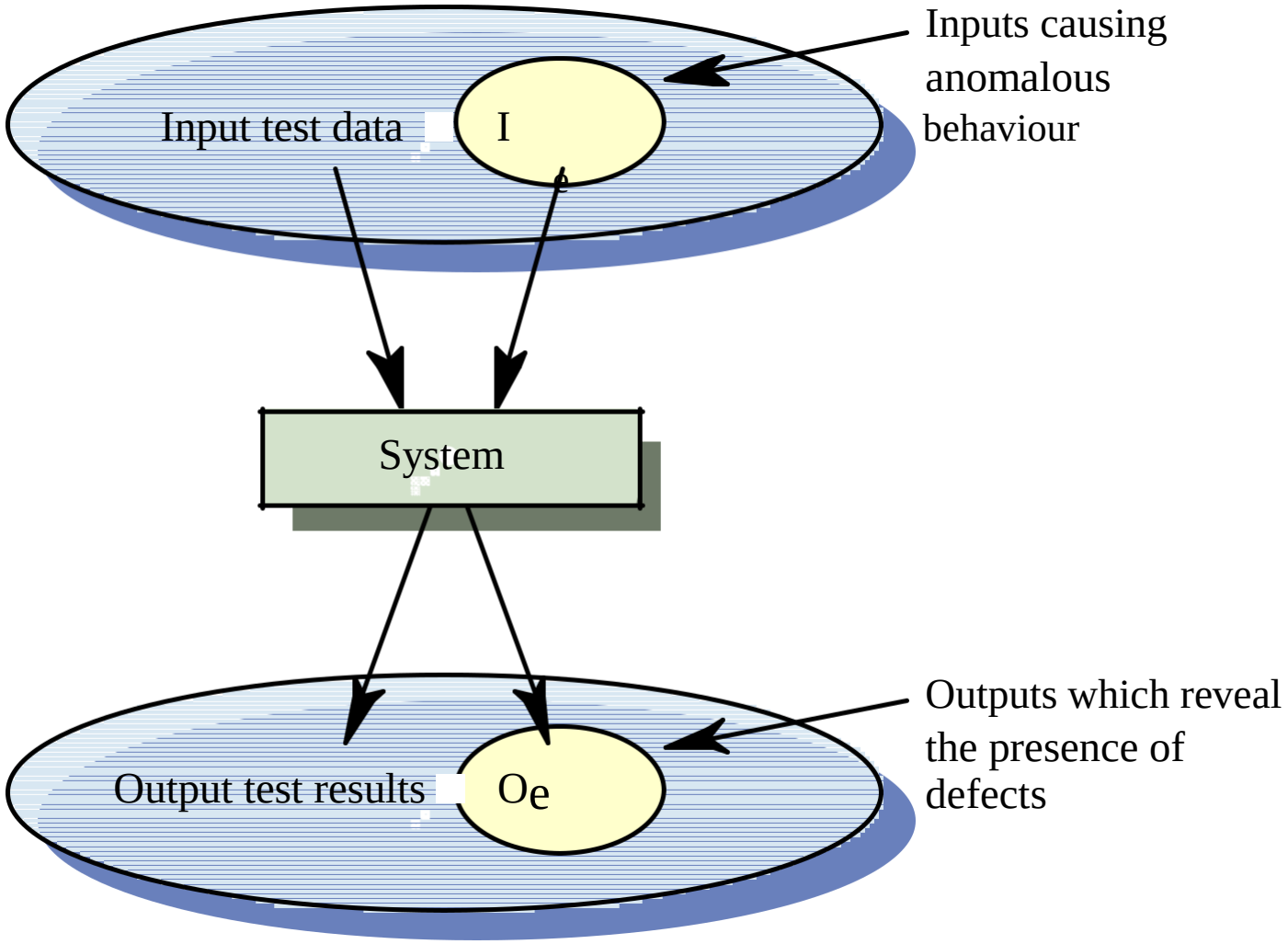
Test methodologies

- Functional (Black box) inspects specified behavior
- Structural (White box) inspects programmed behavior

Black-box testing

- Approach to testing where the tested component is considered as a ‘black-box’
- The program test cases are based on the system specifications
- Formal specification greatly simplifies black-box testing
- Test planning can begin early in the software process

Black-box testing



Black box testing

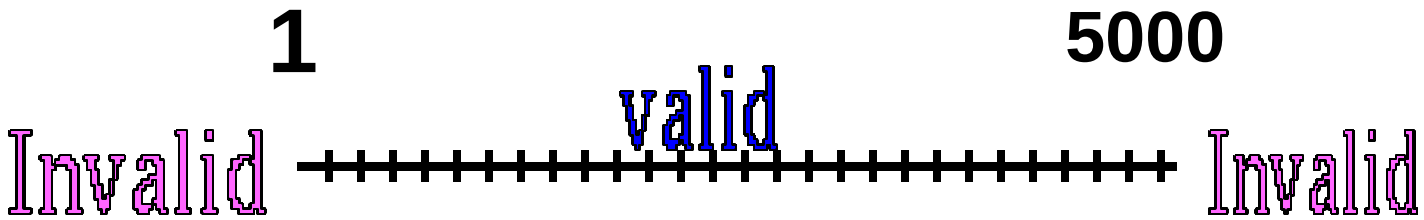
- Equivalence class partitioning
- Boundary value analysis
- Decision Table based testing
  - Cause Effect Graph

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.

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.

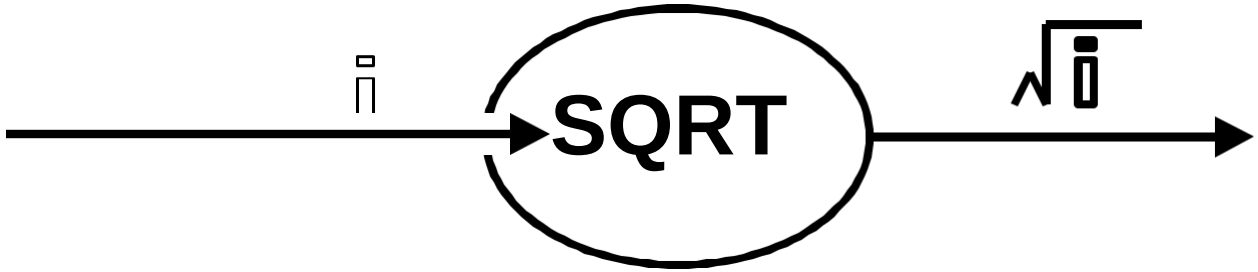


Equivalence Class Partitioning

- 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}.

Cause and Effect Graphs

- Testing would be a lot easier:
  - if we could automatically generate test cases from requirements.
- Work done at IBM:
  - Can requirements specifications be systematically used to design functional test cases?

Cause and Effect Graphs

- Examine the requirements:
  - restate them as logical relation between inputs and outputs.
  - The result is a Boolean graph representing the relationships
    - called a cause-effect graph.

Cause and Effect Graphs

- Convert the graph to a decision table:
  – each column of the decision table corresponds to a test case for functional testing.

Steps to create cause-effect graph

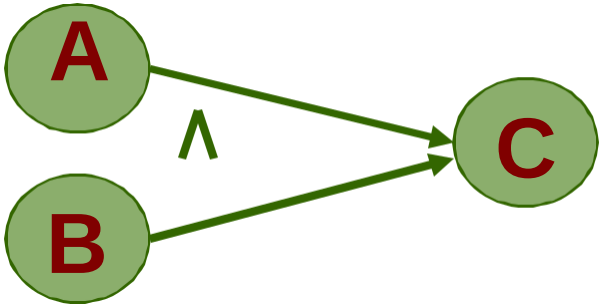
- Study the functional requirements.
- Mark and number all causes and effects.
- Numbered causes and effects:
  - become nodes of the graph.

Steps to create cause-effect graph

- Draw causes on the LHS
- Draw effects on the RHS
- Draw logical relationship between causes and effects
  - as edges in the graph.
- Extra nodes can be added
  - to simplify the graph

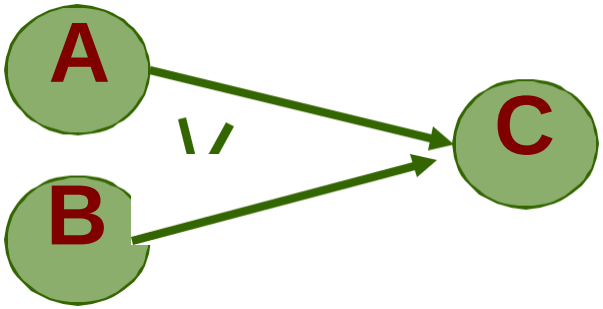
Drawing Cause-Effect Graphs

If A then B

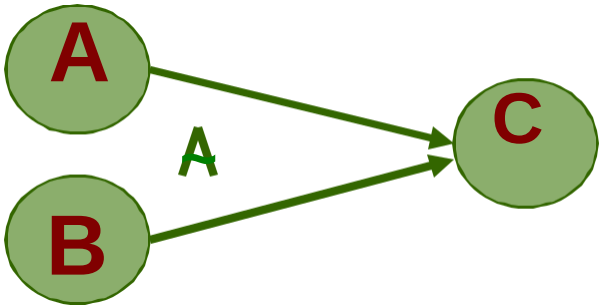


If (A and B)then C

Drawing Cause-Effect Graphs



If (A or B) then C



If (not(A and B)) then C

Cause and Effect Graphs

- Testing would be a lot easier:
  - if we could automatically generate test cases from requirements.
- Work done at IBM:
  - Can requirements specifications be systematically used to design functional test cases?

Cause and Effect Graphs

- Examine the requirements:
  - restate them as logical relation between inputs and outputs.
  - The result is a Boolean graph representing the relationships
    - called a cause-effect graph.

Cause and Effect Graphs

- Convert the graph to a decision table:
  - each column of the decision table corresponds to a test case for functional testing.

Steps to create cause-effect graph

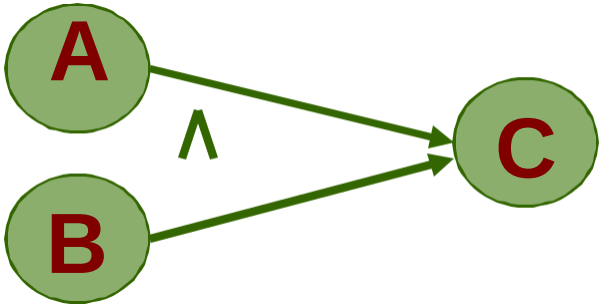
- Study the functional requirements.
- Mark and number all causes and effects.
- Numbered causes and effects:
  - become nodes of the graph.

Steps to create cause-effect graph

- Draw causes on the LHS
- Draw effects on the RHS
- Draw logical relationship between causes and effects
  - as edges in the graph.
- Extra nodes can be added
  - to simplify the graph

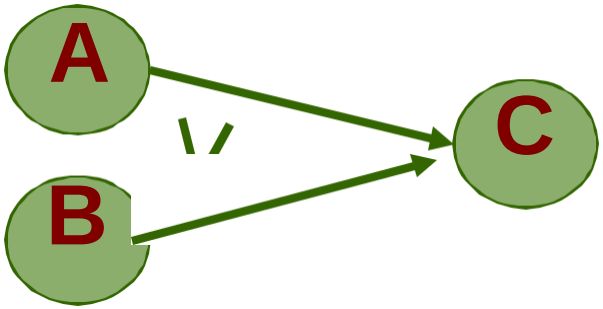
Drawing Cause-Effect Graphs

If A then B

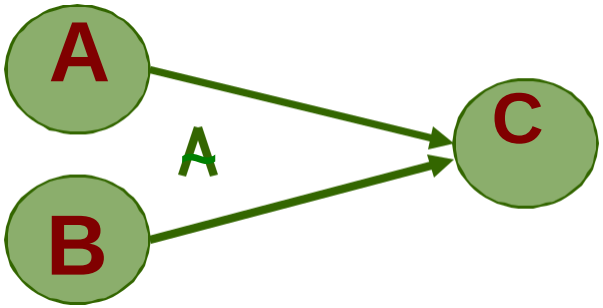


If (A and B)then C

Drawing Cause-Effect Graphs

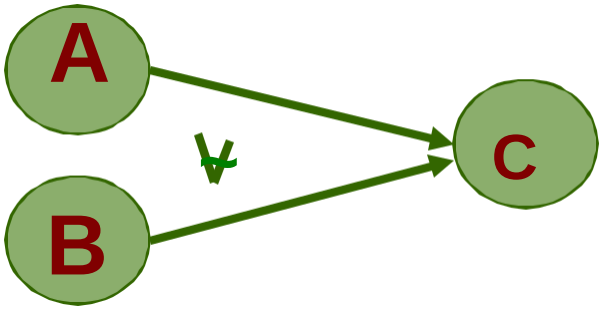


If (A or B) then C



If (not(A and B)) then C

Drawing Cause-Effect Graphs



If (not (A or B))then C

If (not A) then B

Exit Condition:

– if the water content measuring device is not serviceable, go to shutdown mode;else
– if the steaming rate measurement device is not serviceable, go to shutdown mode; else
– if less than three feedpump/feedpump monitor combinations are working correctly, go to shutdown mode; else

...

causes:

- C221 - externally initiated (Either Operator or Instrumentation system)
- C220 - internally initiated
- C202 - operator initiated
- C203 - instrumentation system initiated
- C201 - bad startup
- C200 - operational failure
- C197 - confirmed keystroke entry
- C198 - confirmed "shutnow" message

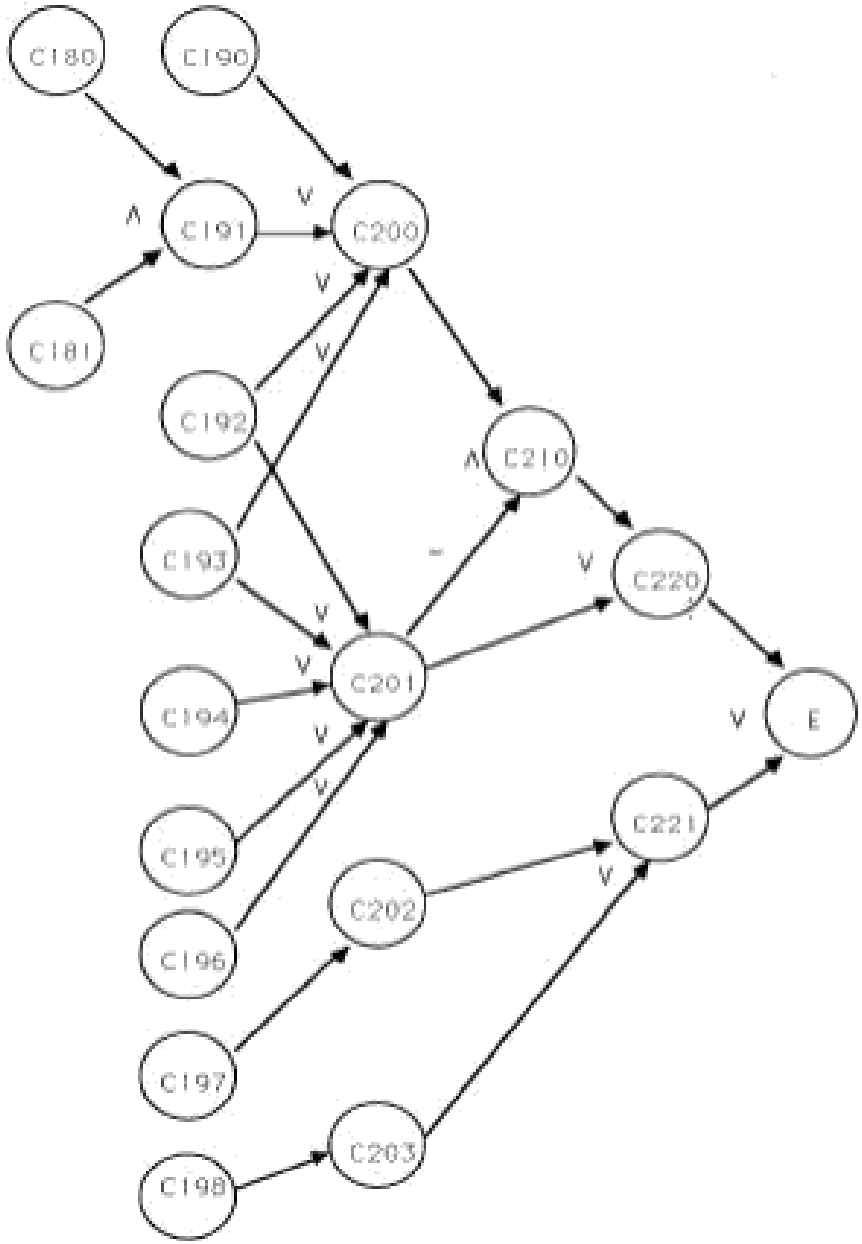
Cont…

- C196 - multiple pumps failure (more than one)
- C195 - water level meter failure during startup
- C194 - steam rate meter failure during startup
- C193 - communication link failure
- C192 - instrumentation system failure
- C191 - C180 and C181

Cont…

- C190 - water level out of range
- C180 - water level meter failure during operation
- C181 - steam rate meter failure during operation
  - Note that some of the causes listed above are used as dummies, and exist only for classification purpose. These causes and their relationships leading to the boiler shutdown are illustrated in the Cause-Effect Graph in Figure 1.

Cause Effect Graph



Decision Table

- Two dimensional mapping of *condition* against *actions* to be performed
  - Conditions evaluate to Boolean
  - Action corresponds to expected activity
- They can be derived from Cause Effect graph too
  - Map cause as condition
  - Map effect as action

Decision Table

Test case design

C_1:x,y,z are sides of a triangle?	N	Y							
C_2:x = y?	--	Y				N			
C_3:x = z?	--	Y		N		Y		N	
C_4:y = z?	--	Y	N	Y	N	Y	N	Y	N
a_1: Not a triangle	X								
a_2: Scalene									X
a_3: Isosceles					X		X	X	
a_4: Equilateral		X							
a_5: Impossible			X	X		X			

Table 3: Decision table for triangle problem

White-Box Testing

- Statement coverage
- Branch coverage
  - Cyclomatic complexity
- Loop testing

Statement Coverage

- Statement coverage methodology:
  - design test cases so that every statement in a program is executed at least once.
- The principal idea:
  - unless a statement is executed, we have no way of knowing if an error exists in that statement

Statement coverage criterion

- Observing that a statement behaves properly for one input value:
  - no guarantee that it will behave correctly for all input values.

Example Euclid's GCD Algorithm

```
•    int f1(int x, int y){
1.   while (x != y){
2.       if (x>y) then
3.            x=x-y;
4.       else y=y-x;
5.    }
6.   return x;          }
```

Euclid's GCD computation algorithm

- By choosing the test set
  {(x=3,y=3),(x=4,y=3), (x=3,y=4)}
  – all statements are executed at least once.

Branch Coverage

- Test cases are designed such that:
  - different branch conditions is given true and false values in turn.
- Branch testing guarantees statement coverage:
  - a stronger testing compared to the statement coverage-based testing.

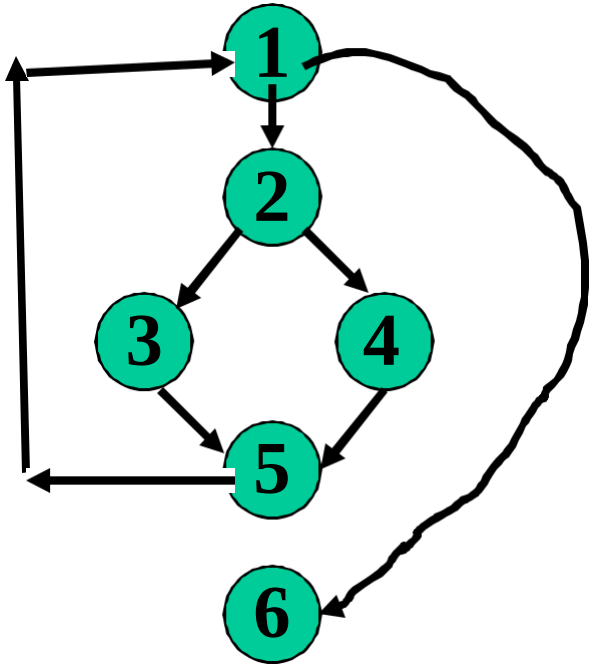
Example

- Test cases for branch coverage can be:
  {(x=3,y=3), (x=4,y=3), (x=3,y=4)}

Example

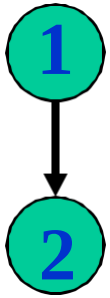
```
int f1(int x,int y){
1.   while (x != y){
2.       if (x>y) then
3.            x=x-y;
4.       else y=y-x;
5.    }
6.    return x;          }
```

Example Control Flow Graph



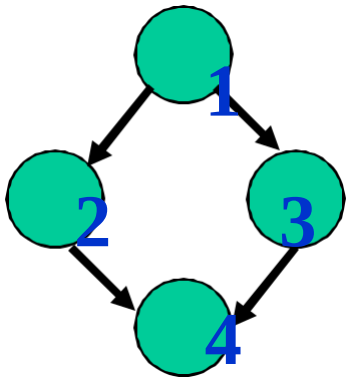
How to draw Control flow graph?

- Sequence:
  - 1 a=5;
  - 2 b=a*b-1;



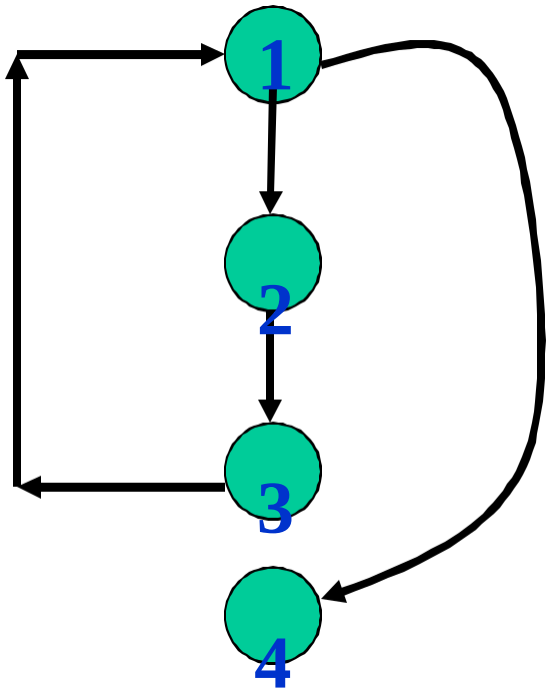
How to draw Control flow graph?

- Selection:
  - 1 if(a>b) then
  - 2 c=3;
  - 3 else c=5;
  - 4 c=c*c;



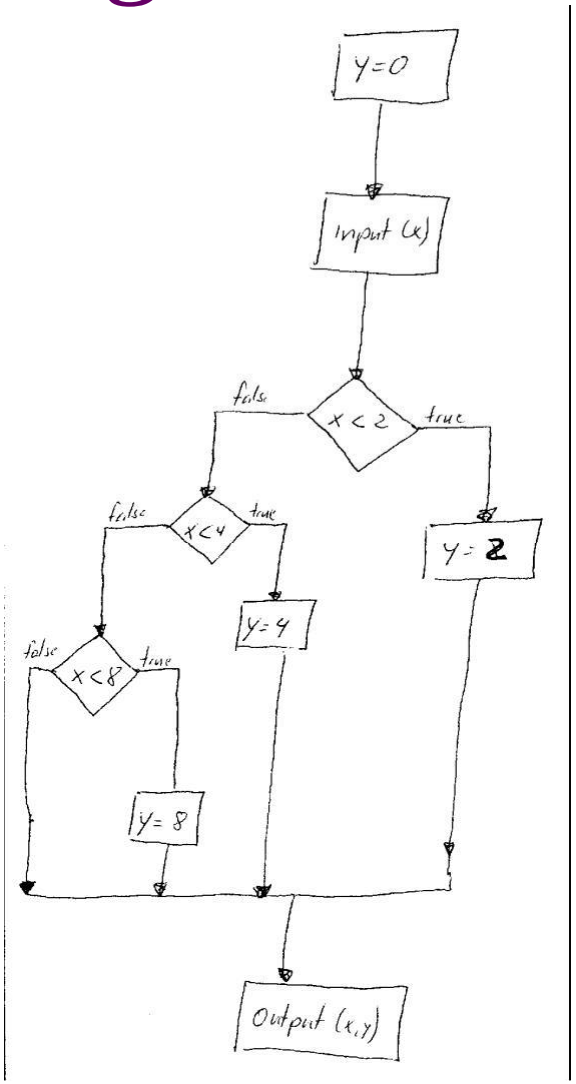
How to draw Control flow graph?

- Iteration:
  - 1 while(a>b){
  - 2 b=b*a;
  - 3 b=b-1;}
  - 4 c=b+d;



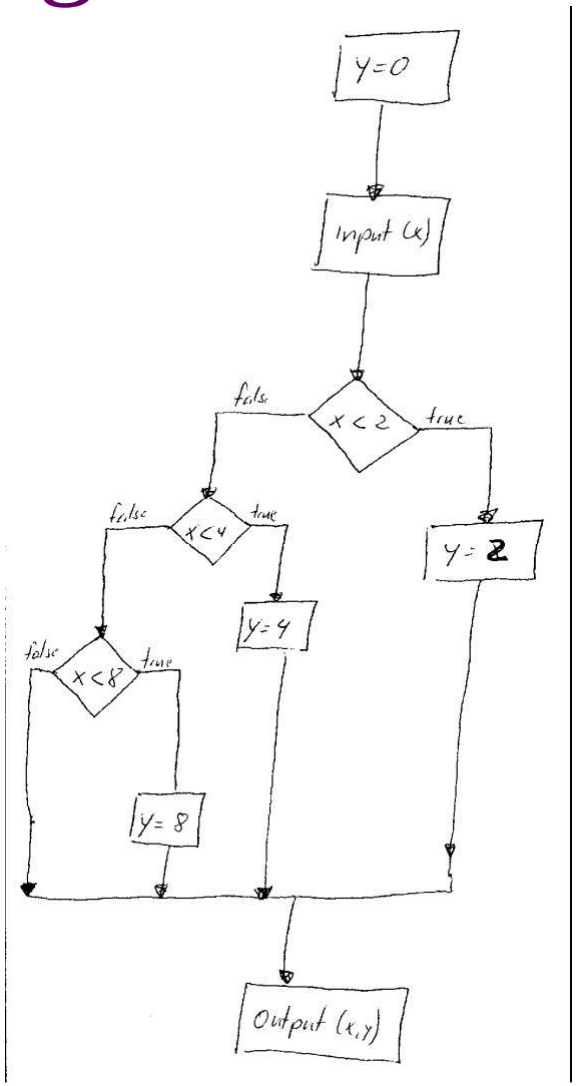
Statement testing

- Every statement should be executed at least once.
- Example:
  - Test #1 : x= 1
  - Test #2 : x = 3
  - Test #3 : x = 7
  - What about x = 9 ?



Branch testing

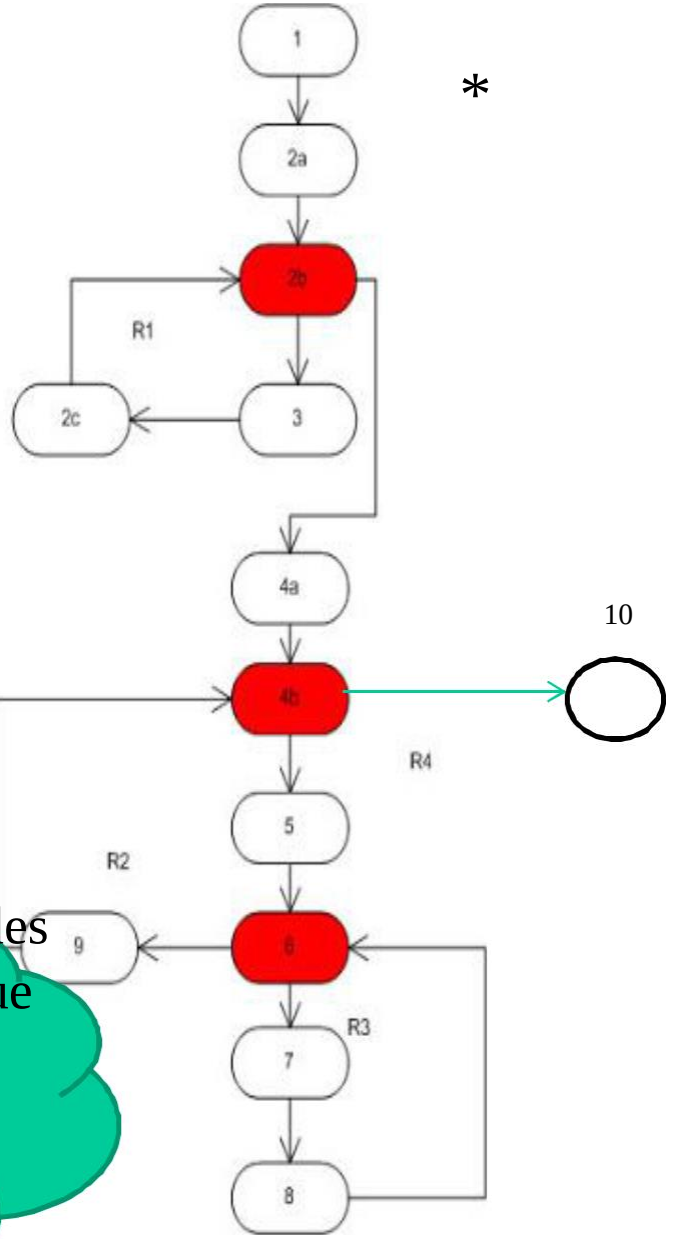
- Every branch in the program should be executed at least once.
- Using statement testing with the 3 test cases, one branch was never executed.
- Test case #4 is necessary with x= 9 as it will check the leftmost branch(It could be the case that leftmost branch has some executable statement). Therefore branch testing covers Statement Testing.



```
insertion_procedure (int a[], int p [],
    int N)
(1){ Int i,j,k;
(2) for ((2a)i=0; (2b)i<=N; (2c)i++)
(3) p[i] = i;
(4) for ((4a)i=2; (4b)i<=N; (4c)i++) {
(5) k=p[i];j=1;
(6) while (a[p[j-1]] > a[k]) {

(7) p[j] = p[j-1];
 (8) j-- }
(9) p[j] = k;}
(10) }
```

*



Note: Control Flow Graph Nodes should be drawn as Circle. *Due to zooming nodes are lil bit stretched.

To calculate Cyclomatic Complexity, One can use one of three methods:

- Count the number of regions on the graph: 4

  V(G) = Enclosed region + 1(Outer Region)

  = 3+1=4

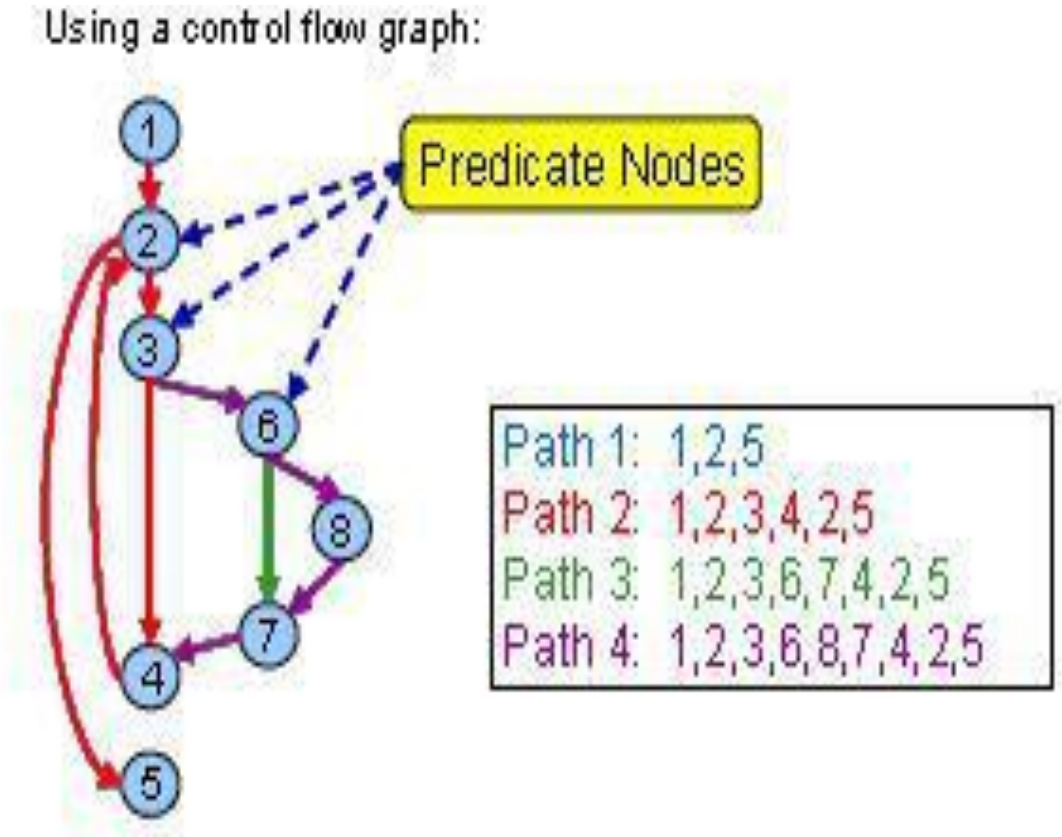
- V(G)= No. of predicates Nodes (Decision Nodes) + 1

  V(G)=3+1=4

- V(G)= No of edges - no. of nodes + 2

  V(G)=15-13+2=4.

Linearly Independent Path



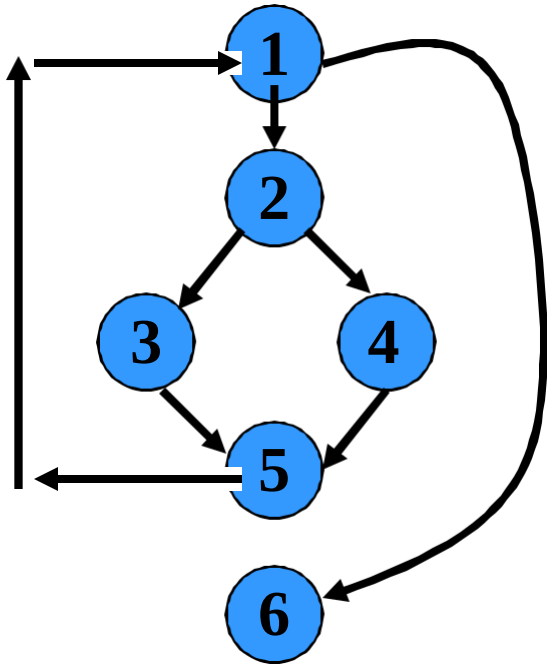
Derivation of Test Cases

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

Example

- int f1(int x,int y){
- 1 while (x != y){
- 2 if (x>y) then
- 3 x=x-y;
- 4 else y=y-x;
- 5 }
- 6 return x; }

Example Control Flow Diagram



Derivation of Test Cases

- Number of independent paths: 3
  - 1,6 test case (x=1, y=1)
  - 1,2,3,5,1,6 test case(x=1, y=2)
  - 1,2,4,5,1,6 test case(x=2, y=1)

Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop m < n
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.
Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one
another then treat each as a simple loop
else* treat as nested loops
endif*
for example, the final loop counter value of loop 1 is used to initialize loop 2.

Example: Flowgraph Complexity

Consider a flow graph given in Fig. 23 and calculate the cyclomatic complexity by all three methods.

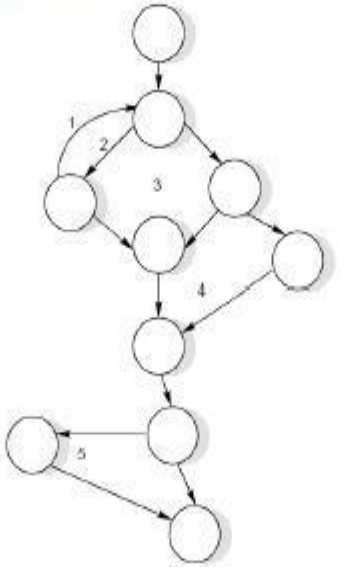


Fig. 23

Complexity

NOTE:Ifuraskedtomake Flow graph of a given Program, DO MAKE A FLOW GRAPH NOT A FLOW CHART

V(G)= No. of Edges - No. of Nodes + 2 x Connected Components I.e. V(G)= e-n+2P or V(G)= 13-10+2x1

V(G)= No. of Predicate Nodes + 1 or 4+1=5

V(G)= No. of flow regions(Inner + outer)=5 i.e
No. of Enclosed Regions + 1 or 4+1=5

Interface testing

- Takes place when modules or sub-systems are integrated to create larger systems (integration testing)
- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces
- Particularly important for object-oriented development as objects are defined by their interfaces

Interfaces types

- Parameter interfaces
  - Data passed from one procedure to another
- Shared memory interfaces
  - Block of memory is shared between procedures
- Procedural interfaces
  - Sub-system encapsulates a set of procedures to be called by other sub-systems

Interface errors

- Interface misuse
  - A calling component calls another component and makes an error the use of its interface e.g. parameters in the wrong order
- Interface purpose misunderstanding
  - A calling component embeds assumptions about the behavior of the called component which are incorrect
- Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which should cause the called component to fail
- Use stress testing to reveal timing problems, especially of multithreading is used.
- If shared memory interfaces are used, test different orders of read/write accesses to the shared memory.

Dynamic stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light
- Stressing the system test failure behavior. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Particularly relevant to distributed systems which can exhibit severe degradation as a network becomes overloaded

Back-to-back dynamic testing

- Present the same tests to different versions of the system and compare outputs. Differing outputs imply potential problems
- Reduces the costs of examining test results. Automatic comparison of outputs.
- Possible when a prototype is available or with regression testing of a new system version

Key points

- Test parts of a system which are commonly used rather than those which are rarely executed
- Equivalence partitions are sets of test cases where the program should behave in an equivalent way
- Black-box testing can be based on the system specification
- Structural testing identifies test cases which cause all paths through the program to be executed

Key points

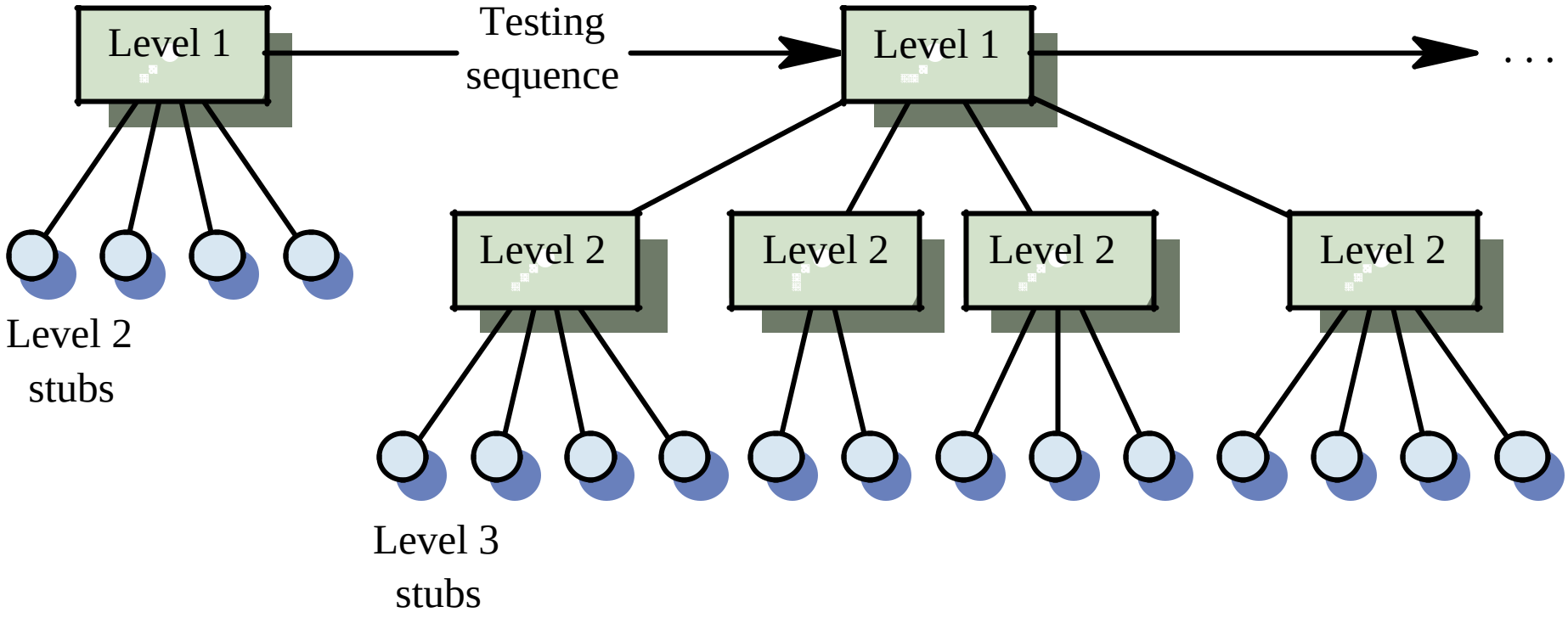
- It is not possible to exercise all path combinations for large, complex systems, especially for integration testing
- Interface defects arise because of specification misreading, misunderstanding, errors or invalid timing assumptions

Integration testing

Integration testing strategies

- Integration testing is about the incremental integration and testing of modules in the system
- Integration testing strategies are ways integrating modules in different orders
- Strategies covered
  - Top-down testing
  - Bottom-up testing
  - Stress testing
  - Back-to-back testing

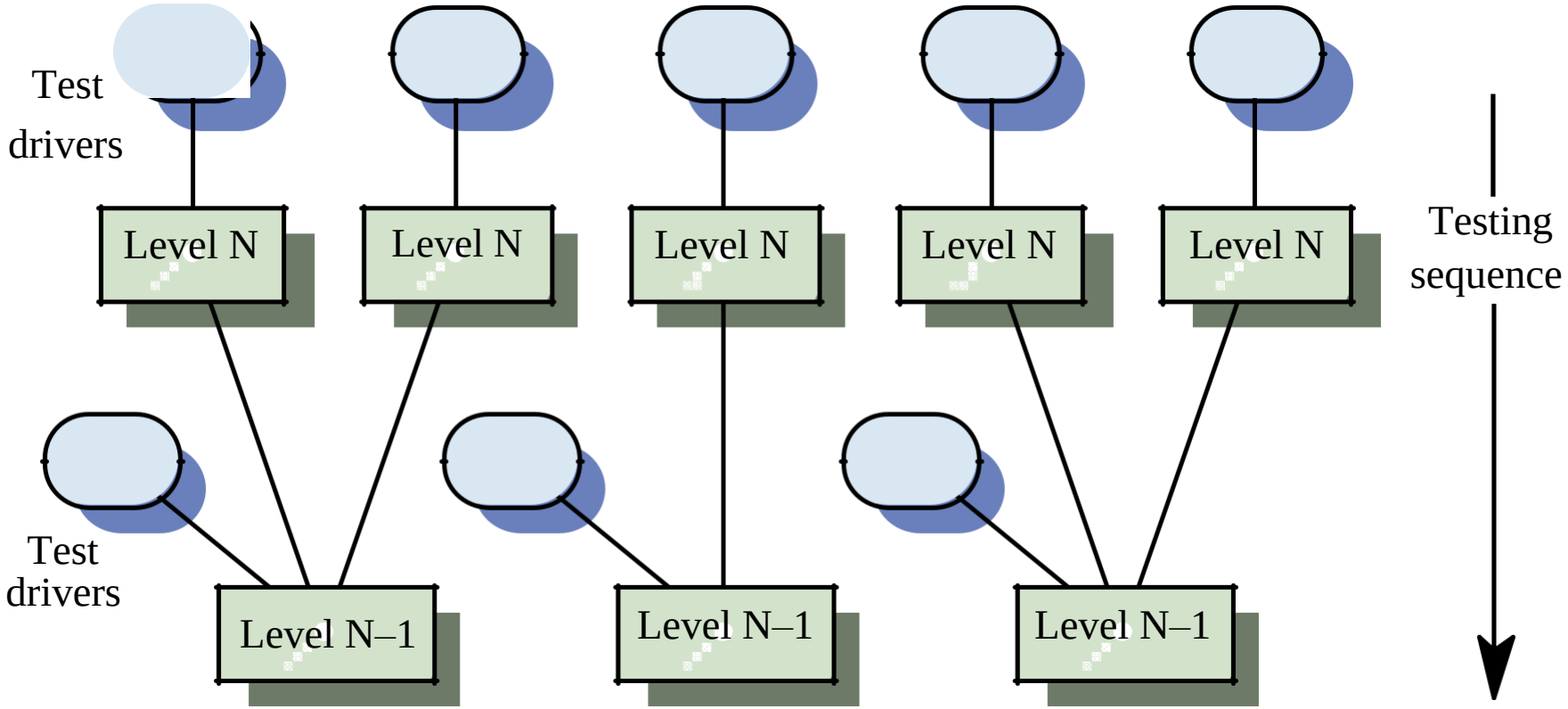
Top-down dynamic testing



Top-down dynamic testing

- Start with the high-levels of a system and work your way downwards
- Testing strategy which is used in conjunction with top-down development
- Finds architecture-related errors
- May need system infrastructure before any testing is possible
- May be difficult to develop program stubs

Bottom-up dynamic testing



Bottom-up dynamic testing

- Necessary for critical infrastructure components
- Start with the lower levels of the system and work upward
- Needs test drivers to be implemented
- Does not find major design problems until late in the process
- Appropriate for object-oriented systems

Key points

- Dynamic and static testing are not the same thing
- Testing is used to establish the presence of defects and to show fitness for purpose
- Testing activities include unit testing, module testing, sub-system testing, integration testing and acceptance testing

Key points

- Testing should be scheduled as part of the planning process. Adequate resources must be made available
- Test plans should be drawn up to guide the testing process
- Testing strategies include top-down testing, bottom-up testing, stress testing and back-to-back testing

System testing

System Testing

- After the whole system has been integrated, it is time to apply tests on the system as a whole.
- This involves testing not components, but the features and qualities of the system as a whole.
- System testing often has to be decomposed in various phases that will incrementally test various aspects of the system's quality:
  - Function testing
  - Performance testing
  - Acceptance testing
  - Installation testing

System testing priorities

- Only complete testing can show a program is free from defects. However, complete testing is impossible
- Tests should exercise a system's capabilities rather than its components
- Testing old capabilities is more important than testing new capabilities
- Testing typical situations is more important than boundary value cases

Function Testing

- The software is thoroughly tested in an environment that simulates the real-life environment of the system.
- E.g. an aircraft flight-control system is provided with input close to that it would receive in normal and abnormal flight conditions
- Compares the system's actual performance with its functional requirements
- Undertaken by a testing team
- Yields a system that is functional, which can be effectively used for further, more elaborated testing.

Performance Testing

- The system's non-functional requirements are tested to and beyond their limits.
  - Volume, Configuration, Security, Recovery tests, etc
- Tests have to be designed to see how the system reacts to overload.
  - Memory consumption, response time, crash, data loss, etc.
- E.g. how does the flight control system react to extreme situations like loss of power, engines, pressurization, or emergency landing, etc.
- Compares the system's actual performance with its non-functional requirements
- Undertaken by a testing team
- Yields a system that is functional and operational even in extreme situations, which can be properly tested by the client in acceptance testing.

Acceptance Testing

- New tests are defined and conducted by the client that will effectively test if the system meets their needs (from their own point of view).
- Might rely to alpha, beta, or back-to-back testing techniques.
- E.g. the flight control system is used by real, experienced pilots in a flight simulator.
- Undertaken by the client, with help from the developers and the testing team
- Yields a system that is functional and operational according to the client's point of view. The system can be installed and used in a real-life environment.

Installation Testing

- After installation at the user site(s), the system is again tested to make sure that it reacts properly to its real-life environment.
- Should concentrate on device interactions, and inter-process communication in the case of distributed systems.
- E.g. the flight control system is installed on board a real aircraft and tested in all kinds of normal and abnormal situations. Physical constraints can affect the system's operation.
- Conducted by the client and the testing team, with help from the development team if needed.
- When this is complete, the system is officially delivered and operational. The maintenance phase is now beginning.

Smoke Testing

- A *smoke test* is a subset of the test cases that is typical representative of the overall test plan.
  - Smoke tests are good for verifying proper deployment or other non invasive changes.
  - **They are also useful for verifying a build is ready to send to test.**
  - Smoke tests are not substitute for actual functional testing.
  - **-Smoke Testing can be used to test at Integration for testing the connections.**

Conclusion

- Verification and Validation (a.k.a testing) is one of the most important and time consuming activities of software development and engineering.
- Determining the cost applied to testing (i.e. when to stop testing) is of prime importance, yet very difficult.
- For critical systems, the cost of testing can be a huge proportion of the cost of the whole system development.

Conclusion

- Testing comes in various phases: reviews for each development phase, unit testing, integration testing, functional testing, performance testing, acceptance testing, and installation testing.
- All these phases must be carefully planned, documented, and automated.
- The design of test cases for all these phases is based on various sources: requirements, specifications, high-level and low-level design, and code.
- A plethora of techniques can be used to automate test case generation.
- Full, 100% system testing is not feasible in most cases.