

	WWW.STUDENTSFOCUS.COM	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for	Software testing fundamentals-internal and external views of testing	
Time:	50 Minutes	
Lesson. No	Unit – IV Lesson No: 1,2/9	

1.Content List: Software testing fundamentals-internal and external views of testing

2.Skill addressed:

- Understand and analysis the Concepts of Software testing fundamentals-internal and external views of testing

3.Objectives of this Lesson Plan:

- To enable students to understand the Concept of Software testing fundamentals-internal and external views of testing

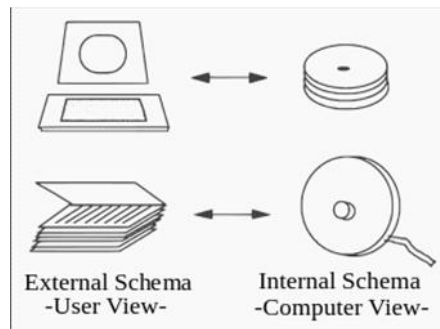
4.Outcome(s):

- Understand And Analyze the basic concepts of Software testing fundamentals-internal and external views of testing

5.Link sheet:

1. What is software testing?
2. What all are the types of software testing?

6.Evocation: (5 Minutes)



Subject introduction (40 Minutes):

7.Lecture Notes

Topics: Characteristics of Testable Software

- Operable
 - The better it works (i.e., better quality), the easier it is to test
- Observable
 - Incorrect output is easily identified; internal errors are automatically detected
- Controllable
 - The states and variables of the software can be controlled directly by the tester
- Decomposable
 - The software is built from independent modules that can be tested independently
- Simple
 - The program should exhibit functional, structural, and code simplicity
- Stable
 - Changes to the software during testing are infrequent and do not invalidate existing tests
- Understandable
 - The architectural design is well understood; documentation is available and organized

Test Characteristics

- A good test has a high probability of finding an error
 - The tester must understand the software and how it might fail
- A good test is not redundant
 - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be -best of breed||

- Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
 - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

Two Unit Testing Techniques

- Black-box testing
 - Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
 - Includes tests that are conducted at the software interface
 - Not concerned with internal logical structure of the software
- White-box testing
 - Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
 - Involves tests that concentrate on close examination of procedural detail
 - Logical paths through the software are tested
 - Test cases exercise specific sets of conditions and loops

What is good test?

A good test has a high probability of Finding an error

- A good test is not redundant.
- A good test should be –best of breed||
- A good test should be neither too simple nor too complex

Internal and External Views

Any engineered product (and most other things) can be tested in one of two ways:

- Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
- Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

Test Case Design

- Objective to uncover error
- Criteria in a complete manner
- Constraint with a minimum of effort and time

8.Textbook:

T1: Roger S. Pressman, –Software Engineering – A practitioner’s Approach||, Sixth Edition, McGraw-Hill International Edition, 2005

9.Application: Testing

	WWW.STUDENTSFOCUS.COM	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for White box testing –basis path testing –control structure testing		
Time:	50 Minutes	
Lesson. No	Unit – IV Lesson No: 2,3/9	

1.Content List: White box testing –basis path testing –control structure testing

2.Skill addressed:

- Understand and analysis the Concepts of White box testing –basis path testing – control structure testing

3.Objectives of this Lesson Plan:

- To enable students to understand the Concept of White box testing –basis path testing –control structure testing

4.Outcome(s):

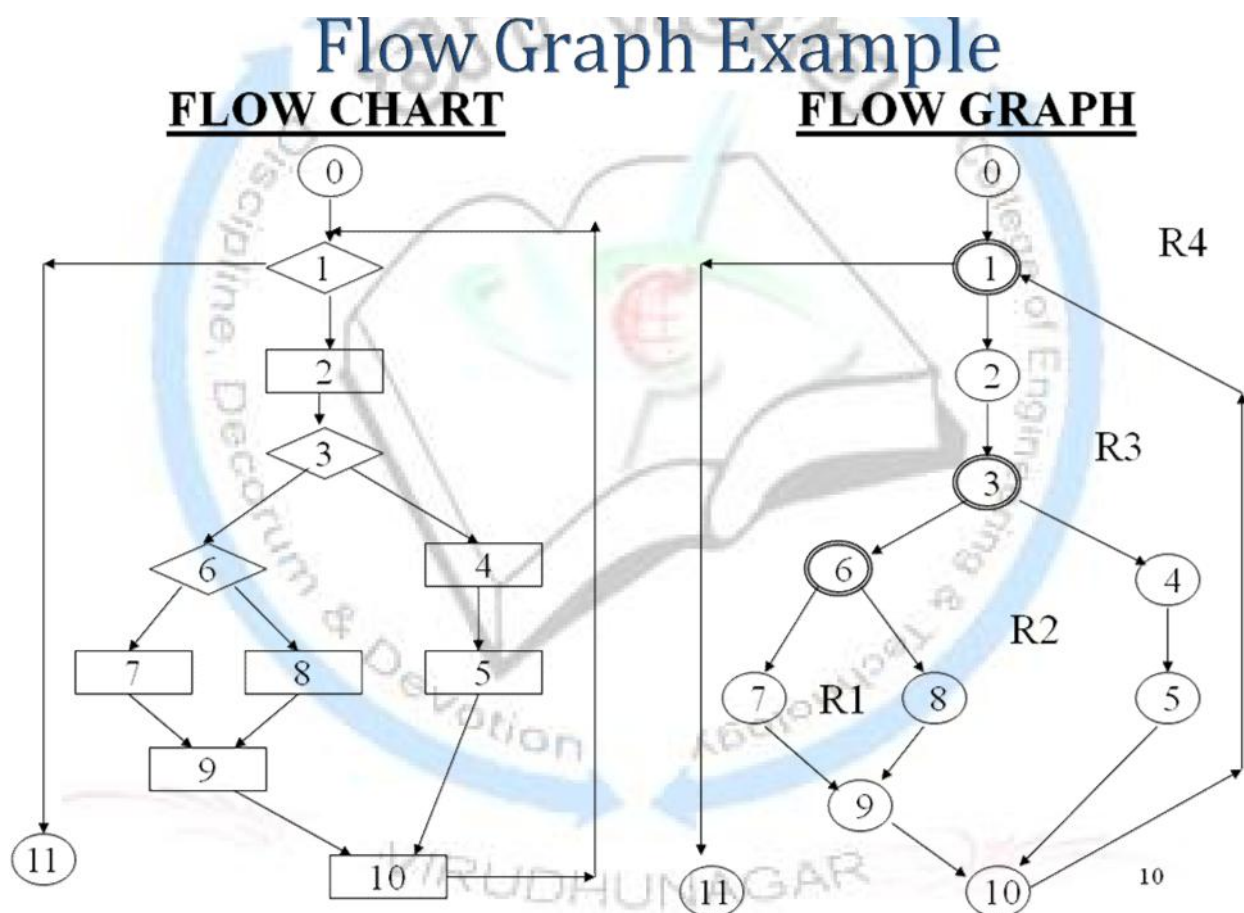
- Understand And Analyse the basic concepts of White box testing –basis path testing – control structure testing

5.Link sheet:

3. What is meant by White box testing?

4. List out the types of white box testing.

6.Evocation: (5 Minutes)



Subject introduction (40 Minutes):

Topics:

- White box testing

White Box Testing

The white box testing is a testing method which is based on close examination of procedural details. Hence it is also called as glass box testing. In white box testing the test cases are derived for

1. Examining all the independent paths within a module.
2. Exercising all the logical paths with their true and false sides.
3. Executing all the loops within their boundaries and within operational bounds.
4. Exercising internal data structures to ensure their validity.

Why to perform white box testing?

There are three main reasons behind performing the white box testing.

1. Programmers may have some incorrect assumptions while designing or implementing some functions. Due to this there are chances of having logical errors in the program. To detect and correct such logical errors procedural details need to be examined.
2. Certain assumptions on flow of control and data may lead programmer to make design errors. To uncover the errors on logical path, white box testing is must.
3. There may be certain typographical errors that remain undetected even after syntax and type checking mechanisms. Such errors can be uncovered during white box testing.

Cyclomatic Complexity

Cyclomatic complexity is a software metric that gives the quantitative measure of logical complexity of the program. The Cyclomatic complexity defines the number of independent paths in the basis set of the program that provides the upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once.

The cyclomatic complexity can be computed by one of the following ways.

1. The number of regions of the flow graph correspond to the cyclomatic complexity.
2. Cyclomatic complexity, $V(G)$, for a flow graph, G , is defined as:

$$V(G) = E - N + 2 ,$$

E - number of flow graph edges,

N - number of flow graph nodes

$$3. V(G) = P + 1$$

where P is the number of predicate nodes contained in the flow graph G .

Structural Testing

- The structural testing is sometime called as white-box testing.
- In structural testing derivation of test cases is according to program structure. Hence knowledge of the program is used to identify additional test cases.
- Objective of structural testing is to exercise all program statements.

Condition Testing

To test the logical conditions in the program module the condition testing is used. This condition can be a Boolean condition or a relational expression. The condition is incorrect in following situations.

- i) Boolean operator is incorrect, missing or extra.

ii) Boolean variable is incorrect.

iii) Boolean parenthesis may be missing, incorrect or extra.

iv) Error in relational operator.

v) Error in arithmetic expression.

The condition testing focuses on each testing condition in the program.

- The branch testing is a condition testing strategy in which for a compound condition each and every true or false branches are tested.
- The domain testing is a testing strategy in which relational expression can be tested using three or four tests.

Basis Path Testing

- White-box testing technique proposed by Tom McCabe
- Enables the test case designer to derive a logical complexity measure of a procedural design
- Uses this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing

Flow Graph Notation

- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
 - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
 - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is an arrow representing flow of control in a specific direction
 - An edge must start and terminate at a node
 - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too

Independent Program Paths

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along at least one edge that has not been traversed before by a previous path
- Basis set for flow graph on previous slide
 - Path 1: 0-1-11
 - Path 2: 0-1-2-3-4-5-10-1-11
 - Path 3: 0-1-2-3-6-8-9-10-1-11
 - Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity

Deriving the Basis Set and Test Cases

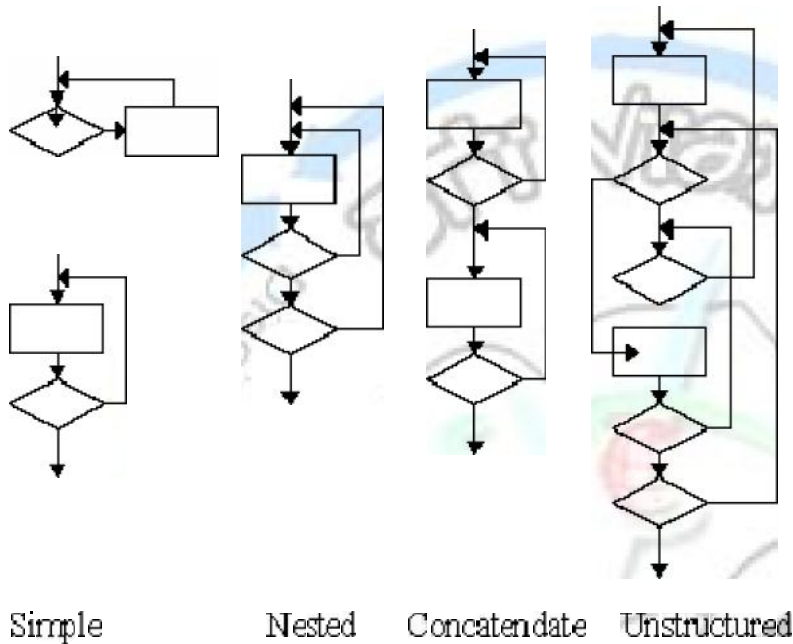
- 1) Using the design or code as a foundation, draw a corresponding flow graph
- 2) Determine the cyclomatic complexity of the resultant flow graph
- 3) Determine a basis set of linearly independent paths

- 4) Prepare test cases that will force execution of each path in the basis set

Loops are fundamental to many algorithms and need thorough testing.

There are four different classes of loops: simple, concatenated, nested, and unstructured.

Examples:



Create a set of tests that force the following situations:

- **Simple Loops**, where n is the maximum number of allowable passes through the loop.
 - Skip loop entirely
 - Only one pass through loop
 - Two passes through loop
 - m passes through loop where $m < n$.
 - $(n-1)$, n , and $(n+1)$ passes through the loop.
- **Nested Loops**
 - Start with inner loop. Set all other loops to minimum values.
 - Conduct simple loop testing on inner loop.
 - Work outwards
 - Continue until all loops tested.
- **Concatenated Loops**
 - If independent loops, use simple loop testing.
 - If dependent, treat as nested loops.
- **Unstructured loops**

- Don't test - redesign.



```
public class loopdemo
{
    private int[] numbers = {5,-3,8,-12,4,1,-20,6,2,10};

    /** Compute total of numItems positive numbers in the array
     * @param numItems how many items to total, maximum of 10.
     */
    public int findTotal(int numItems)
    {
        int total = 0;
        if (numItems <= 10)
        {
            for (int count=0; count < numItems; count = count + 1)
            {
                if (numbers[count] > 0)
                {
                    total = total + numbers[count];
                }
            }
        }
        return total;
    }
}

public void testOne()
{
    loopdemo app = new loopdemo();
    assertEquals(0, app.findTotal(0));
    assertEquals(5, app.findTotal(1));
    assertEquals(5, app.findTotal(2));
    assertEquals(17, app.findTotal(5));
    assertEquals(26, app.findTotal(9));
    assertEquals(36, app.findTotal(10));
    assertEquals(0, app.findTotal(11));
}
```

8.Textbook:

T1: Roger S. Pressman, -Software Engineering – A practitioner's Approach||, Sixth Edition, McGraw-Hill International Edition, 2005

	<div>Sri Vidya College of Engineering &Technology</div> <div>Department of Information Technology</div>	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for Black box testing, Regression testing		
Time:	50 Minutes	
Lesson. No	Unit – IV Lesson No: 5,6/9	

1. Content List: Black box testing, Regression testing

2. Skill addressed:

- Understand and analysis the Concepts of Black box testing, Regression testing

3. Objectives of this Lesson Plan:

- To enable students to understand the Concept of Black box testing, Regression testing

4. Outcome(s):

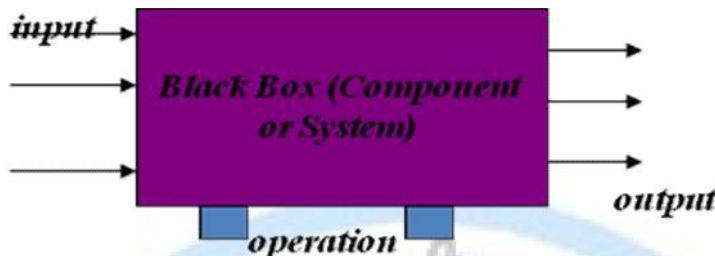
- Understand And Analyze the basic concepts of Black box testing, Regression testing

5. Link sheet:

5. What is meant by black box testing?

6. Define Boundary value analysis.
7. What is meant by regression testing?

6. Evocation: (5 Minutes)



Subject introduction (40 Minutes):

Topics:

- Black box testing
- Equivalence Class Guidelines:
- Boundary Value Analysis
- Regression testing

7. Lecture Notes

Black-Box Testing:

Graph-Based Testing – 1:

- Black-box methods based on the nature of the relationships (links) among the program objects (nodes), test cases are designed to traverse the entire graph
- Transaction flow testing
 - nodes represent steps in some transaction and links represent logical connections between steps that need to be validated
- Finite state modeling
 - nodes represent user observable states of the software and links represent state transitions

Graph-Based Testing – 2:

- Data flow modeling
 - nodes are data objects and links are transformations of one data object to another data object
- Timing modeling
 - nodes are program objects and links are sequential connections between these objects
 - link weights are required execution times

Equivalence Partitioning:

- Black-box technique that divides the input domain into classes of data from which test cases can be derived
- An ideal test case uncovers a class of errors that might require many arbitrary test cases to be executed before a general error is observed

Equivalence Class Guidelines:

- If input condition specifies a range, one valid and two invalid equivalence classes are defined

- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
- If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined
- If an input condition is Boolean, one valid and one invalid equivalence class is defined
- Boundary Value Analysis - 1
- Black-box technique
 - focuses on the boundaries of the input domain rather than its center
- Guidelines:
 - If input condition specifies a range bounded by values a and b, test cases should include a and b, values just above and just below a and b
 - If an input condition specifies a number of values, test cases should exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values

For example

- Area code: input condition. Boolean - the area code mayor may not be present.
- Input condition, range - value defined between 200 and 700.
- Password: input condition, Boolean - a password mayor may not be present.
- Input condition, value - seven character string.
- Command: input condition, set - containing commands noted before.

Boundary Value Analysis

- A boundary value analysis is a testing technique in which the elements at the edge of the domain are selected and tested.
- Instead of focusing on input conditions only, the test cases from output domain are also derived.
- Boundary value analysis is a test case design technique that complements equivalence partitioning technique.

Guidelines for boundary value analysis technique are

1. If the input condition specified the range bounded by values x and y, then test cases should be designed with values x and y. Also test cases should be with the values above and below x and y.
2. If input condition specifies the number of values then the test cases should be designed with minimum and maximum values as well as with the values that are just above and below the maximum and minimum should be tested.
3. If the output condition specified the range bounded by values x and y, then test cases should be designed with values x and y. Also test cases should be with the values above and below x and y.
4. If output condition specifies the number of values then the test cases should be designed with minimum and maximum values as well as with the values that are just above and below the maximum and minimum should be tested.
5. If the internal program data structures specify such boundaries then the test cases must be designed such that the values at the boundaries of data structure can be tested.

Boundary Value Analysis – 2

1. Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maximum output reports
2. If internal program data structures have boundaries (e.g. size limitations), be certain to test the boundaries

For example

Integer D with input condition [-2, 10],

test values: -2, 10, 11, -1, 0

If input condition specifies a number values, test cases should developed to exercise the minimum and maximum .numbers. Values just above and below this min and max should be tested.

Enumerate data E with input condition: {2, 7, 100, 102}

Test values: 2, 102, -1, 200, 7

Comparison Testing:

- Black-box testing for safety critical systems in which independently developed implementations of redundant systems are tested for conformance to specifications
- Often equivalence class partitioning is used to develop a common set of test cases for each implementation

Orthogonal Array Testing – 1:

- Black-box technique that enables the design of a reasonably small set of test cases that provide maximum test coverage
- Focus is on categories of faulty logic likely to be present in the software component (without examining the code)

Orthogonal Array Testing – 2:

- Priorities for assessing tests using an orthogonal array
 - Detect and isolate all single mode faults
 - Detect all double mode faults
 - Multimode faults

Regression Testing:



The selective retesting of a software system that has been modified to ensure that any bugs have been fixed and that no other previously working functions have failed as a result of the reparations and that newly added features have not created problems with previous versions of the software. Also referred to as verification testing, regression testing is initiated after a programmer has attempted to fix a recognized problem or has added source code to a program that may have inadvertently introduced errors. It is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity.

Regression Testing:

- Regression test suit contains 3 different classes of test cases
 - Representative sample of existing test cases is used to exercise all software functions.
 - Additional test cases focusing software functions likely to be affected by the change.
 - Tests cases that focus on the changed software components.

8.Textbook:

.T1: Roger S. Pressman, –Software Engineering – A practitioner's Approach||, Sixth Edition, McGraw-Hill International Edition, 2005

	Sri Vidya College of Engineering & Technology Department of Information Technology	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for	unit testing, integration, validation testing, System testing and debugging	
Time:	50 Minutes	
Lesson. No	Unit – IV Lesson No: 7,8/9	

1.Content List: unit testing, integration, validation testing, System testing and debugging

2.Skill addressed:

- Understand and analysis the Concepts of unit testing, integration, validation testing, System testing and debugging

3.Objectives of this Lesson Plan:

- To enable students to understand the Concept of unit testing, integration, validation testing, System testing and debugging

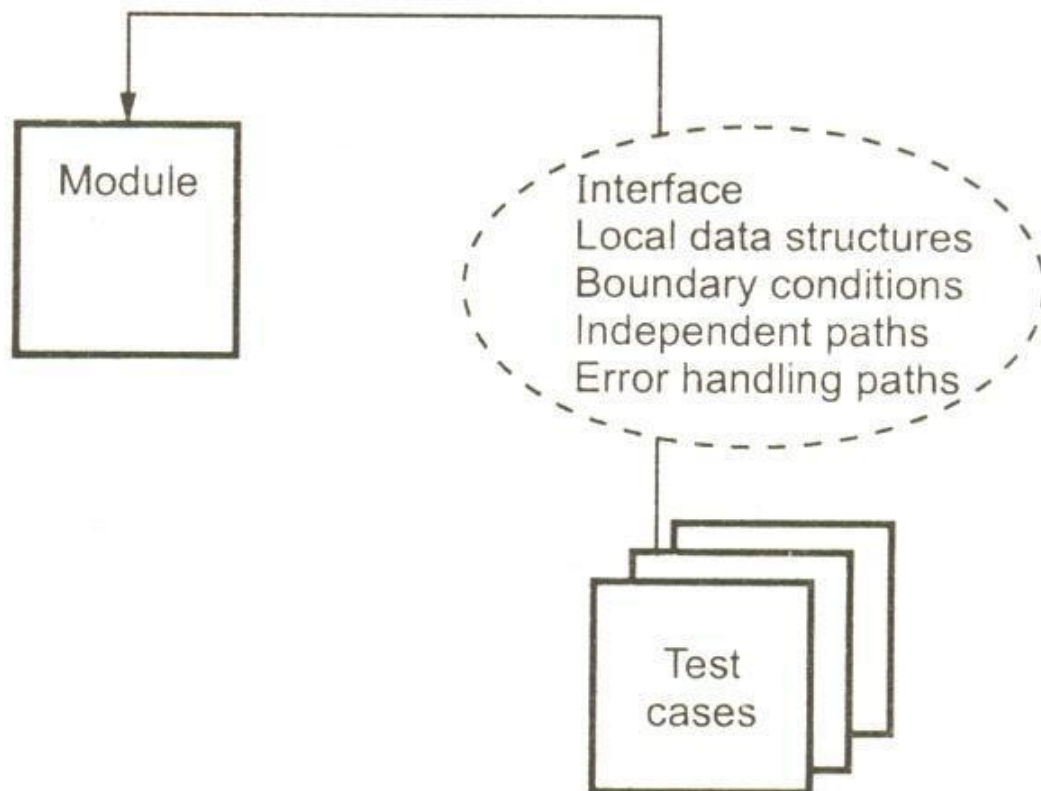
4.Outcome(s):

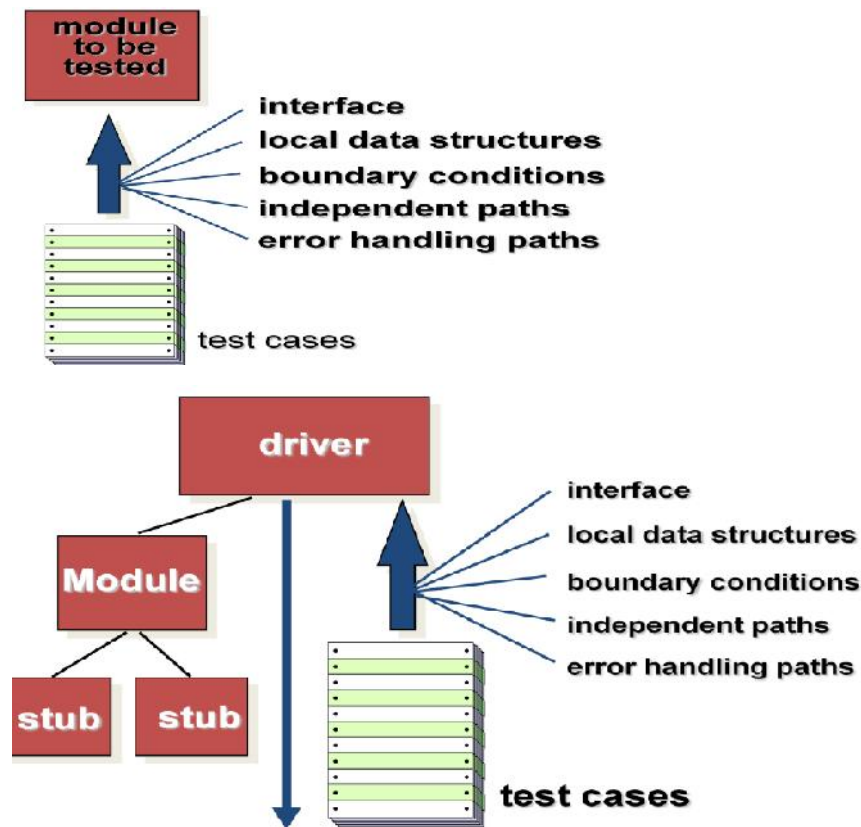
- Understand And Analyse the basic concepts of unit testing, integration, validation testing, System testing and debugging

5.Link sheet:

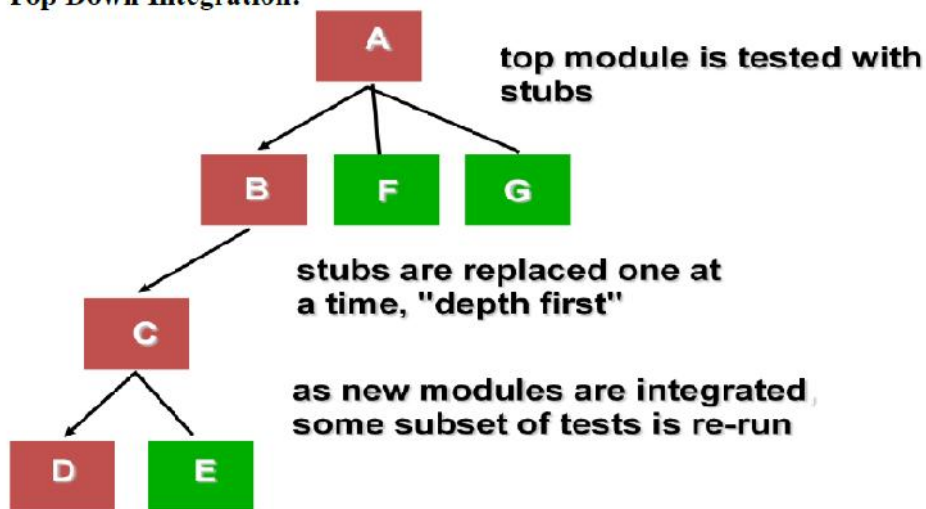
1. Define unit testing
2. What all are the types of integration testing?
3. Define system testing.

6.Evocation: (5 Minutes)

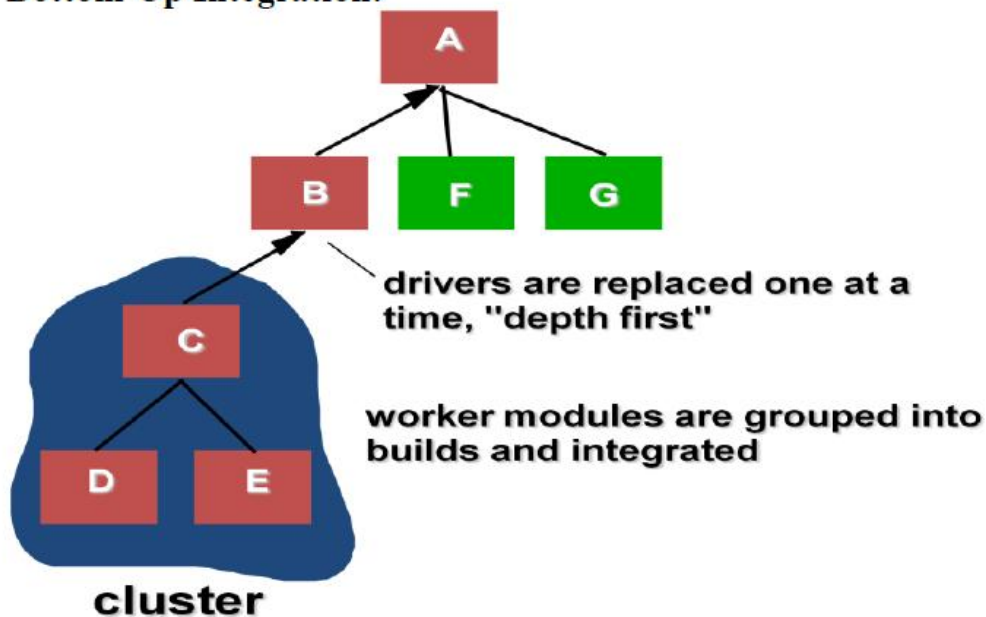




Top Down Integration:



Bottom-Up Integration:



Subject introduction (40 Minutes):

Topics:

7.Lecture Notes

Unit Testing Details:

- Interfaces tested for proper information flow.
- Local data are examined to ensure that integrity is maintained.
- Boundary conditions are tested.
- Basis path testing should be used.
- All error handling paths should be tested.
- Drivers and/or stubs need to be developed to test incomplete software.

Unit Testing:

In unit testing the individual components are tested independently to ensure their quality. The focus is to uncover the errors in design and implementation.

The various tests that are conducted during the unit test are described as below -

1. Module interfaces are tested for proper information flow in and out of the program.
2. Local data are examined to ensure that integrity is maintained.
3. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
4. All the basis (independent) paths are tested for ensuring that all statements in the module have been executed only once.
5. All error handling paths should be tested.

Integration Testing:

- Bottom - up testing (test harness).
- Top - down testing (stubs).
- Regression Testing.
- Smoke Testing

Top-Down Integration Testing:

- Main program used as a test driver and stubs are substitutes for components directly subordinate to it.
- Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
- Tests are conducted as each component is integrated.
- On completion of each set of tests and other stub is replaced with a real component.

Einstein College of Engineering

- Regression testing may be used to ensure that new errors not introduced.

Bottom-Up Integration:

Bottom-Up Integration Testing:

- Low level components are combined in clusters that perform a specific software function.
- A driver (control program) is written to coordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program structure.

Regression Testing:

- The selective retesting of a software system that has been modified to ensure that any bugs have been fixed and that no other previously working functions have failed as a result of the reparations and that newly added features have not created problems with previous versions of the software. Also referred to as verification testing, regression testing is initiated after a programmer has attempted to fix a recognized problem or has added source code to a program that may have inadvertently introduced errors. It is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity.

Regression Testing:

- Regression test suit contains 3 different classes of test cases
 - Representative sample of existing test cases is used to exercise all software functions.
 - Additional test cases focusing software functions likely to be affected by the change.
 - Tests cases that focus on the changed software components.

Smoke Testing:

- Software components already translated into code are integrated into a build.
- A series of tests designed to expose errors that will keep the build from performing its functions are created.
- The build is integrated with the other builds and the entire product is smoke tested daily using either top-down or bottom integration.

Validation Testing

- Ensure that each function or performance characteristic conforms to its specification.
- Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.

Acceptance Testing:

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha test

version of the complete software is tested by customer under the supervision of the developer at the developer's site

- **Beta test**

version of the complete software is tested by customer at his or her own site without the developer being present

System Testing:

- **Recovery testing**

checks system's ability to recover from failures

- **Security testing**

verifies that system protection mechanism prevents improper penetration or data alteration

- **Stress testing**

program is checked to see how well it deals with abnormal resource demands

- **Performance testing**

tests the run-time performance of software

Performance Testing:

- Stress test.

- Volume test.

- Configuration test (hardware & software).

- Compatibility.

- Regression tests.

Security tests.

- Timing tests.

- Environmental tests.

- Quality tests.

- Recovery tests.

- Maintenance tests.

- Documentation tests.

- Human factors tests.

Testing Life Cycle:

- Establish test objectives.

- Design criteria (review criteria).

- Correct.

- Feasible.

- Coverage.

- Demonstrate functionality.

Writing test cases.

- Testing test cases.

- Execute test cases.

- Evaluate test results.

Testing Tools:

- Simulators.

- Monitors.

- Analyzers.

- Test data generators.

Document Each Test Case:

- Requirement tested.
- Facet / feature / path tested.
- Person & date.
- Tools & code needed.
- Test data & instructions.
- Expected results.
- Actual test results & analysis
- Correction, schedule, and signoff.



Debugging:

- Debugging (removal of a defect) occurs as a consequence of successful testing.
- Some people better at debugging than others.
- Is the cause of the bug reproduced in another part of the program?
- What —next bug|| might be introduced by the fix that is being proposed?
- What could have been done to prevent this bug in the first place?

8.Textbook:

T1: Roger S. Pressman, –Software Engineering – A practitioner's Approach||, Sixth Edition, McGraw-Hill International Edition, 2005

9.Application: Testing

	Sri Vidya College of Engineering &Technology Department of Information Technology	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for Software Implementation techniques		
Time:	50 Minutes	
Lesson. No	Unit – IV Lesson No: 9/9	

1.Content List: Software Implementation techniques

2.Skill addressed:

- Understand and analysis the Concepts of Software Implementation techniques

3.Objectives of this Lesson Plan:

- To enable students to understand the Concept of Software Implementation techniques

4.Outcome(s):

- Understand And Analyse the basic concepts of Software Implementation techniques

5.Link sheet:

6.Evocation: (5 Minutes)

Subject introduction (40 Minutes):

Topics: Software Implementation techniques

7.Lecture Notes

Software Implementation techniques

- Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.
- Software Implementation Techniques include process and thread scheduling, synchronization and concurrency primitives, file management, memory management, performance, networking facilities, and user interfaces. Software Implementation Techniques is designed to facilitate determining what is required to implement a specific operating system function.

Procedural programming

Procedural programming can sometimes be used as a synonym for imperative programming (specifying the steps the program must take to reach the desired state), but can also refer (as in this article) to a programming paradigm, derived from structured programming, based upon the concept of the procedure call. Procedures, also known as routines, subroutines, methods, or functions (not to be confused with mathematical functions, but similar to those used in functional programming) simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself. Some good examples of procedural programs are the Linux Kernel, GIT, Apache Server, and Quake III Arena.

Object-oriented programming

An object-oriented program may thus be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent 'machine' with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to 'carry their own operators around with them' (or at least "inherit" them from a similar object or class). In the conventional model, the data and operations on the data don't have a tight, formal association.

functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.

In practice, the difference between a mathematical function and the notion of a "function" used in imperative programming is that imperative functions can have side effects, changing the value of already calculated computations. Because of this they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program. Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result $f(x)$ both times. Eliminating side effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming. JavaScript, one of the most widely employed languages today, incorporates functional programming capabilities.

Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, which can be traced at least as far back as John McCarthy's [1958] advice-taker proposal, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmer, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently.

Oracle's Application Implementation Method

AIM provides with an integrated set of templates, procedures, PowerPoint presentations, spreadsheets, and project plans for implementing the applications. AIM was such a success,

Oracle created a subset of the templates, called it AIM Advantage, and made it available as a product to customers and other consulting firms. Since its initial release, AIM has been revised and improved several times with new templates and methods.

AIM Is a Six-Phase Method Because the Oracle ERP Applications are software modules buy from a vendor, different implementation methods are used than the techniques used for custom developed systems. AIM has six major phases:

Definition phase: During this phase, you plan the project, determine business objectives, and verify the feasibility of the project for given time, resource, and budget limits.

Operations Analysis phase: Includes documents business requirements, gaps in the software (which can lead to customizations), and system architecture requirements. Results of the analysis should provide a proposal for future business processes, a technical architecture model, an application architecture model, workarounds for application gaps, performance testing models, and a transition strategy to migrate to the new systems. Another task that can begin in this phase is mapping of legacy data to Oracle Application APIs or open interfaces—data conversion.

Solution Design phase—Used to create designs for solutions that meet future business requirements and processes. The design of your future organization comes alive during this phase as customizations and module configurations are finalized.

Build phase—During this phase of AIM, coding and testing of customizations, enhancements, interfaces, and data conversions happens. In addition, one or more conference room pilots test the integrated enterprise system. The results of the build phase should be a working, tested business system solution.

Transition phase—During this phase, the project team delivers the finished solution to the enterprise. End-user training and support, management of change, and data conversions are major activities of this phase.

Production phase—Starts when the system goes live. Technical people work to stabilize and maintain the system under full transaction loads. Users and the implementation team begin a series of refinements to minimize unfavorable impacts and realize the business objectives identified in the definition phase.

Rapid Implementations

In the late 1990s as Y2K approached, customers demanded and consulting firms discovered faster ways to implement packaged software applications. The rapid implementation became possible for certain types of customers. The events that converged in the late 1990s to provide faster implementations include the following:

- Many smaller companies couldn't afford the big ERP project. If the software vendors and consulting firms were going to sell to the —middle market|| companies, they had to develop more efficient methods.
- Many dotcoms needed a financial infrastructure; ERP applications filled the need, and rapid implementation methods provided the way.
- The functionality of the software improved a lot, many gaps were eliminated, and more companies could implement with fewer customizations.
- After the big, complex companies implemented their ERP systems, the typical implementation became less difficult.
- The number of skilled consultants and project managers increased significantly.
- Other software vendors started packaging preprogrammed integration points to the Oracle ERP modules.

Phased Implementations

Phased implementations seek to break up the work of an ERP implementation project. This technique can make the system more manageable and reduce risks, and costs in some cases, to the enterprise. In the mid-1990s, 4 or 5 was about the maximum number of application modules that could be launched into production at one time. If you bought 12 or 13 applications, there would be a financial phase that would be followed by phases for the distribution and manufacturing applications. As implementation techniques improved and Y2K pressures grew in the late 1990s, more and more companies started launching most of their applications at the same time. This method became known as the big-bang approach. Now, each company selects a phased or big-bang approach based on its individual requirements.

Another approach to phasing can be employed by companies with business units at multiple sites. With this technique, one business unit is used as a template, and all applications are completely implemented in an initial phase lasting 10–14 months. Then, other sites implement the applications in cookie-cutter fashion. The cookie-cutter phases are focused on end-user training and the differences that a site has from the prototype site. The cookie-cutter phase can be as short as 9–12 weeks, and these phases can be conducted at several sites simultaneously. For your reference, we participated in an efficient project where 13 applications were implemented big bang-style in July at the Chicago site after about 8 months work. A site in Malaysia went live in October. The Ireland site started up in November. After a holiday break, the Atlanta business unit went live in February, and the final site in China started using the applications in April. Implementing thirteen application modules at five sites in four countries in sixteen months was pretty impressive.

Case Studies Illustrating Implementation Techniques

Some practical examples from the real world might help to illustrate some of the principles and techniques of various software implementation methods. These case studies are composites from about 60 implementation projects we have observed during the past 9 years.

Big companies often have a horrible time resolving issues and deciding on configuration parameters because there is so much money involved and each of many sites might want to control decisions about what it considers its critical success factors. For example, we once saw a large company argue for over two months about the chart of accounts structure, while eight consultants from two consulting firms tried to referee among the feuding operating units. Another large company labored for more than six months to unify a master customer list for a centralized receivables and decentralized order entry system.

Transition activities at large companies need special attention. Training end users can be a logistical challenge and can require considerable planning. For example, if you have 800 users to train and each user needs an average of three classes of two hours each and you have one month, how many classrooms and instructors do you need? Another example is that loading data

Small companies have other problems when creating an implementation team. Occasionally, the small company tries to put clerical employees on the team and they have problems with issue resolution or some of the ERP concepts. In another case, one small company didn't create the position of project manager. Each department worked on its own modules and ignored the integration points, testing, and requirements of other users. When Y2K deadlines forced the system startup, results were disastrous with a cost impact that doubled the cost of the entire project.

- Project team members at small companies sometimes have a hard time relating to the cost of the implementation. We once worked with a company where the project manager (who was also the database administrator) advised me within the first hour of

our meeting that he thought consulting charges of \$3/minute were outrageous, and he couldn't rationalize how we could possibly make such a contribution. We agreed a consultant could not contribute \$3 in value each and every minute to his project. However, when I told him we would be able to save him \$10,000/week and make the difference between success and failure, he realized we should get to work.

- Because the small company might be relatively simple to implement and the technical staff might be inexperienced with the database and software, it is possible that the technical staff will be on the critical path of the project. If the database administrator can't learn how to handle the production database by the time the users are ready to go live, you might need to hire some temporary help to enable the users to keep to the schedule. In addition, we often see small companies with just a single database administrator who might be working 60 or more hours per week. They feel they can afford to have more DBAs as employees, but they don't know how to establish the right ratio of support staff to user requirements. These companies can burn out a DBA quickly and then have to deal with the problem of replacing an important skill.

8.Textbook:

T1: Roger S. Pressman, -Software Engineering – A practitioner's Approach||, Sixth Edition, McGraw-Hill International Edition, 2005

9.Application: Testing