MODULE 3

IMPLEMENTATION AND TESTING

1. Discuss the various ways of identifying various ways of identifying object classes in object-oriented systems? .

1. Use a grammatical analysis of a natural language description of the system to be constructed.

2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations such as offices, organizational units such as companies, and so on.

Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn.

2. Explain the importance of design patterns

Pattern → a description of the problem and the essence of its solution, so that the solution may be reused in different settings.

• not a detailed specification.

• It is a description of accumulated wisdom and experience, a well-tried solution to a common problem.

Patterns are a way of reusing the knowledge and experience of other designers.

• Design patterns are usually associated with object-oriented design.

• Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality.

• General principle of encapsulating experience in a pattern → one that is equally applicable to any kind of software design.

4 essential elements of design patterns as per Gang of Four's book on patterns:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and

   their responsibilities.
4. A statement of the consequences—the results and trade-offs—of applying the

   pattern.

Advantages:

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

3. Explain about Open-source licensing? Compare GPL, LGPL and BSD

A fundamental principle of open-source development is that source code should be freely available.

Legally, the developer of the code owns the code. They can place restrictions on how it is used by including legally binding conditions in an open-source software license

Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source..

The open-source approach is one of several business models for software.

Most open-source licenses are variants of one of three general models:

1. The GNU General Public License (GPL).

This is a so-called reciprocal license that simplistically means that if you use open-source software that is licensed under the GPL license, then you must make that software open source.

2. The GNU Lesser General Public License (LGPL).

This is a variant of the GPL license where you can write components that link to open-source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.

3. The Berkley Standard Distribution (BSD) License.

This is a nonreciprocal license, which means you are not obliged to re-publish any changes or modifications made to open-source code. You can include the code in proprietary systems that are sold. If you use open-source components, you must acknowledge the original creator of the code. eg. The MIT license .

4. What is meant by a code walkthrough? What are some of the important types of errors checked during code walkthrough?

5. Differentiate between code walk through and code inspection?

Code Walkthrough

 We present the code and accompanying documentation to the review team, and the team comments on their correctness.

 During walkthrough, we lead and control the discussion. The atmosphere is informal and the focus of attention is on the code, not the coder.

 Although Supervisory personnel may be present, walkthrough has no influence on the performance appraisal, consistent with the general intent of testing, finding faults, not fixing them.

Code Inspection

 Similar to Code walkthrough, but is more formal. In an inspection, review team checks the code and documentation against a prepared list of concerns.

 For eg: the team may examine the definition and use of data type and structures to see if their use is consistent with the design and with standards and procedures. The team can review algorithms and computations for their correctness and efficiency. Interfaces also checked. The team may estimate the code's performance characteristics in terms of memory usage or processing speed.

6. How can you compute the cyclomatic complexity of a program? How is cyclomatic complexity useful in program testing?

Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors. It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module.
Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand.
Complexity is a software metric that given 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 atleast once.
There are three methods of computing Cyclomatic complexities.

- **Method 1:** Total number of regions in the flow graph is a Cyclomatic complexity.
- **Method 2**: The Cyclomatic complexity, V (G) for a flow graph G can be defined as V (G) = E - N + 2 Where: E is total number of edges in the flow graph. N is the total number of nodes in the flow graph.
- **Method 3**: The Cyclomatic complexity V (G) for a flow graph G can be defined as V (G) = P + 1 Where: P is the total number of predicate nodes contained in the flow G.

7. What is black box testing? Explain the different types of black box testing.strategies. For a software that computes the square root of an input integer that can assume values in the range of 0 and 1000. Determine the equivalence class test suite.

Black-box testing, also called behavioral testing or functional testing focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program

 Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors .

Black box testing strategies

Equivalence partitioning divides the input domain into classes of data that are likely to exercise a specific software function.
Boundary value analysis probes the program's ability to handle data at the limits of acceptability.
Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters.
Model-based testing uses elements of the requirements model to test the behavior of an application.

- A program reads an input value in the range of 1 and 5000:
  - computes the square root of the input number
- There are three equivalence classes:
  - the set of negative integers,
  - set of integers in the range of 1 and 5000,
  - integers larger than 5000.
- The test suite must include:
  - representatives from each of the three equivalence classes:
  - a possible test suite can be:{-5,500,6000}

10. Discuss software Testing strategies (refer diagram and notes for explanation)

<u>Software Testing Strategy—The Big Picture</u>

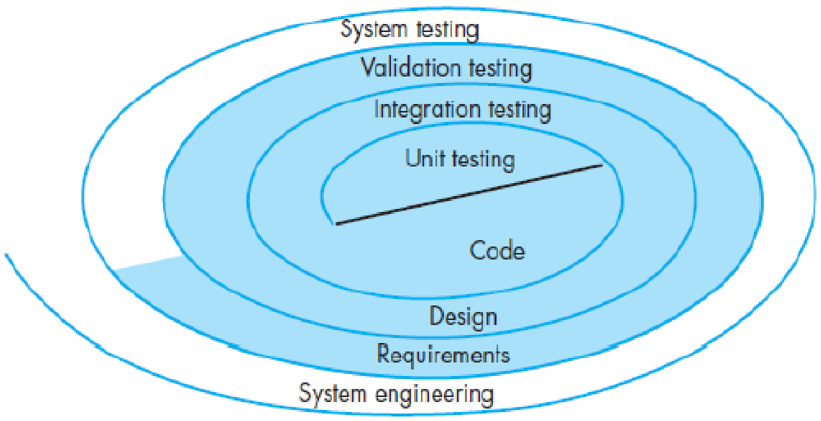


Figure Testing Strategy

Unit Testing

Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code.

 Testing progresses by moving outward along the spiral to ***integration testing,*** where the focus is on design and the construction of the software architecture.

 Taking another turn outward on the spiral, you encounter ***validation testing,*** where requirements established as part of requirements modeling are validated against the software that has been constructed.

 Finally, you arrive at ***system testing,* where** the software and other system elements are tested as a whole. To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.

 Initially, tests focus each component individually, ensuring that it functions properly as a unit. Hence, the name ***unit testing****.* Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.

 Next, components must be assembled or integrated to form the complete software package. *Integration testing* addresses the issues associated with the dual problems of verification and program construction. Testcase design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.

 After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated. *Validation testing* provides final assurance that software meets all functional, behavioral, and performance requirements.

Integration testing

- Top down and bottom up
- Regression Testing
- Smoke Testing

Validation Testing

- Alpha and Beta testing

System testing .

- Recovery Testing
- Security Testing
- Stress Testing
- Performance testing
- Deployment testing

11. Explain Topdown and Bottom up integration testing ? (refer notes for more explanation)*Top-down integration testing* is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).

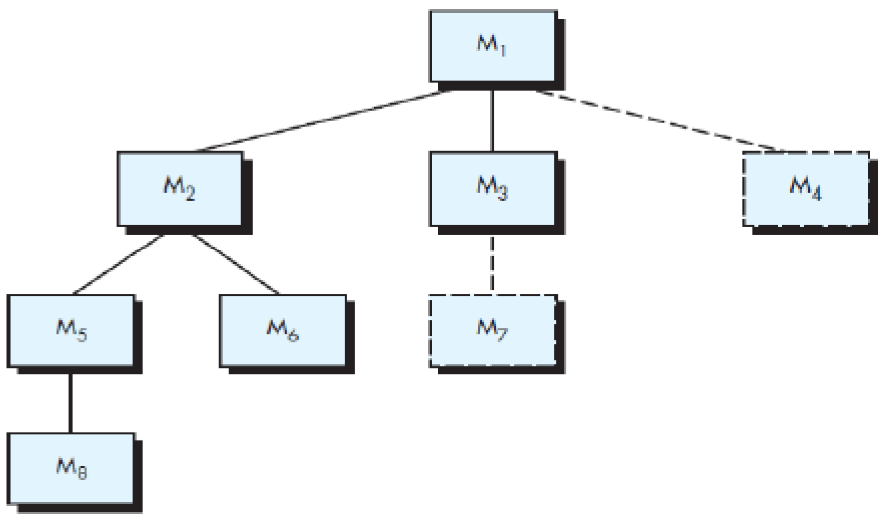
 Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depthfirst or breadth-first manner. Referring to Figure below ,***depth-first integration*** integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.

 For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated.

 Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving acrossthe structure horizontally. From the figure, components M2, M3, and M4 wouldbe integrated first. The next control level, M5, M6, and so on, follows.

 The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actualcomponents.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the realcomponent.
5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.



Bottom-Up Integration. *Bottom-up integration testing,* as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
2. A *driver* (a control program for testing) is written to coordinate test-case input and output.
3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure

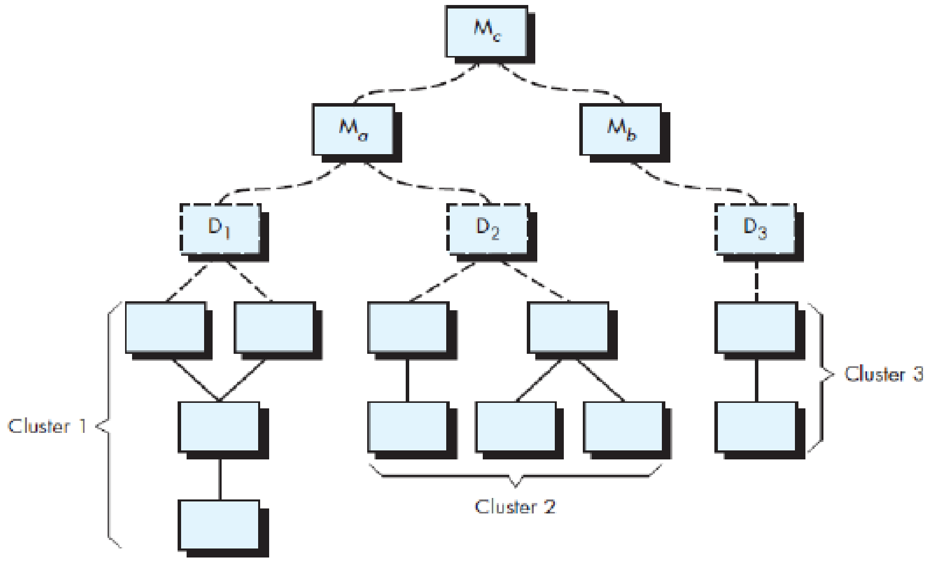


Figure : Bottom up approach

11. Explain basis path testing?

- Basis path testing is a white-box testing technique which enables the test-case designer to derive a logical complexity measure of a procedural design and use 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.

  1. Draw a control graph (to determine different program paths)
  2. Calculate Cyclomatic complexity (metrics to determine the number of independent 3.paths)
  4. Find a basis set of paths.
  5. Generate test cases to exercise each path.

A.1.1 Flow Graph Notation

The flow graph depicts logical control flow using the notation illustrated in Figure

FIGURE 23.1 Flow graph notation

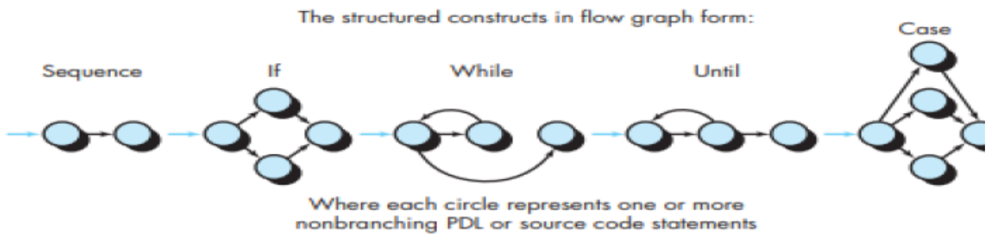
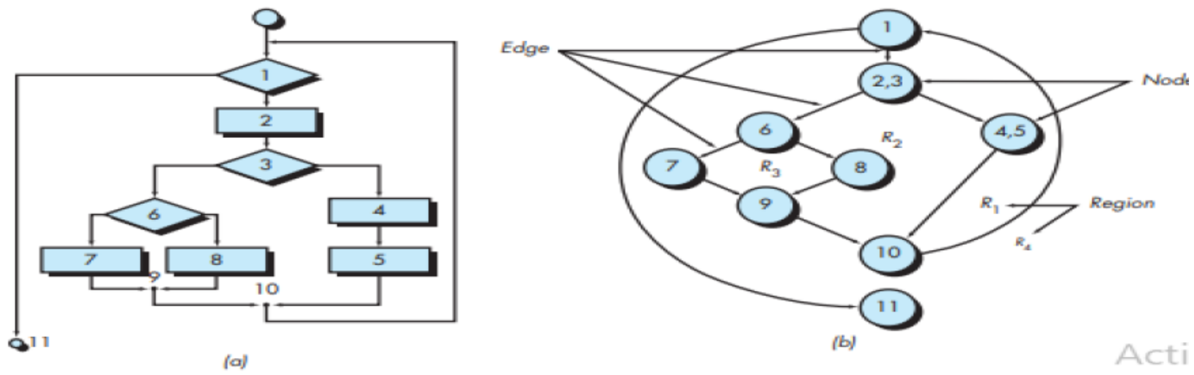

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

FIGURE 23.2 (a) Flowchart and (b) flow graph



- To illustrate the use of a flow graph, consider the procedural design representation in Figure 23.2a . Here, a flowchart is used to depict program control structure.
- Figure 23.2bmaps the flowchart into a corresponding flow
- Referring to Figure 23.2b, each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node. The arrows on the w graph, called edges or links, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct)
- Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.

12. How Black box testing differ from White box testing

S.No	Black Box Testing	White Box Testing
1	The main objective of this testing is to test the Functionality / Behavior of the application.	The main objective is to test the infrastructure of the application.

S.No	Black Box Testing	White Box Testing
2	This can be performed by a tester without any coding knowledge of the AUT (Application Under Test).	Tester should have the knowledge of internal structure and how it works.
3	Testing can be performed only using the GUI.	Testing can be done at an early stage before the GUI gets ready.
4	This testing cannot cover all possible inputs.	This testing is more thorough as it can test each path.
5	Some test techniques include Boundary Value Analysis, Equivalence Partitioning, Error Guessing etc.	Some testing techniques include Conditional Testing, Data Flow Testing, Loop Testing etc.
6	Test cases should be written based on the Requirement Specification.	Test cases should be written based on the Detailed Design Document.
7	Test cases will have more details about input conditions, test steps, expected results and test data.	Test cases will be simple with the details of the technical concepts like statements, code coverage etc.
8	This is performed by professional Software Testers.	This is the responsibility of the Software Developers.
9	Programming and implementation knowledge is not required.	Programming and implementation knowledge is required.
10	Mainly used in higher level testing like Acceptance Testing, System Testing etc.	Is mainly used in the lower levels of testing like Unit Testing and Integration Testing.
11	This is less time consuming and exhaustive.	This is more time consuming and exhaustive.
12	Test data will have wide possibilities so it will be tough to identify the correct data.	It is easy to identify the test data as only a specific part of the functionality is focused at a time.
13	Main focus of the tester is on how the application is working.	Main focus will be on how the application is built.
14	Test coverage is less as it cannot create test data for all scenarios.	Almost all the paths/application flow are covered as it is easy to test in parts.

S.No	Black Box Testing	White Box Testing
15	Code related errors cannot be identified or technical errors cannot be identified.	Helps to identify the hidden errors and helps in optimizing code.
16	Defects are identified once the basic code is developed.	Early defect detection is possible.

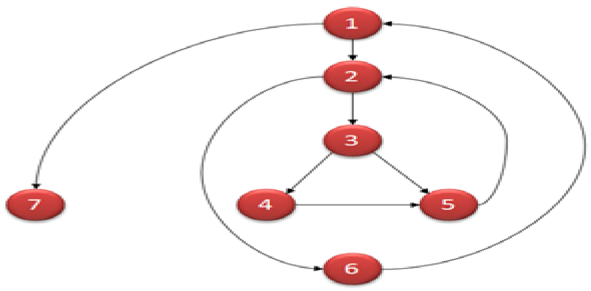
13. Consider the program given below, construct the flow graph and calculate the cyclomatic complexity .

```
i = 0;
n=4; //N-Number of nodes present in the graph
while (i<n-1) do
j = i + 1;
while (j<n) do
if A[i]<A[j] then
swap(A[i], A[j]);
end do;
i=i+1;
end do
```

Mathematical representation:

Mathematically, it is set of independent paths through the graph diagram. The Code complexity of the program can be defined using the formula –

V(G) = E - N + 2



Computing mathematically,

V(G) = 9 – 7 + 2 = 4

V(G) = 3 + 1 = 4 (Condition nodes are 1,2 and 3 nodes)

Basis Set – A set of possible execution path of a program

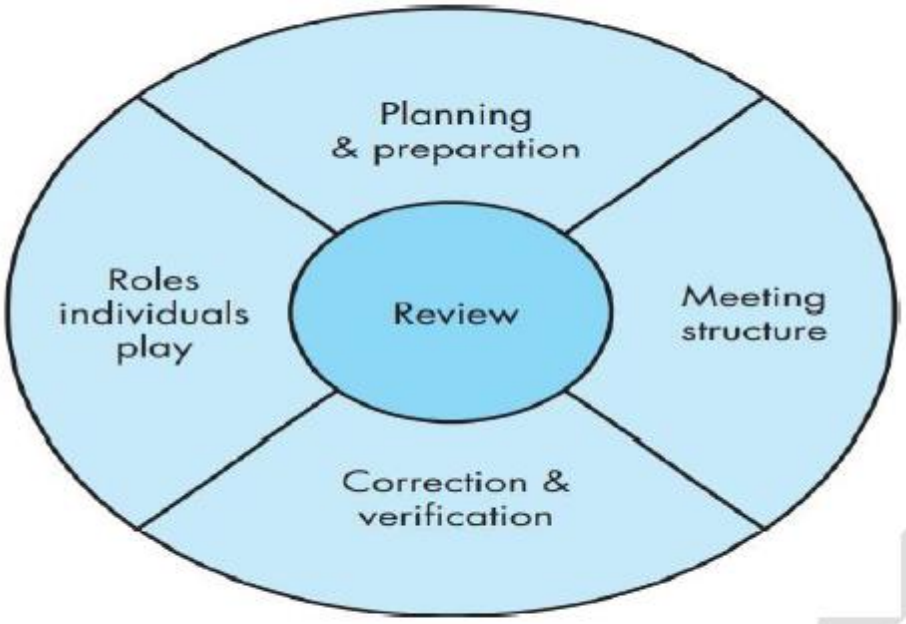
1, 7

1, 2, 6, 1, 7

1, 2, 3, 4, 5, 2, 6, 1, 7

1, 2, 3, 5, 2, 6, 1, 7

13. Explain about formal and informal reviews ?



formal technical review (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are:
(1) to uncover errors in function, logic, or implementation for any representation of the software;
(2) to verify that the software under review meets its requirements;
(3) to ensure that the software has been represented according to predefined standards;
(4) to achieve software that is developed in a uniform manner;
(5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software.

The FTR is actually a class of reviews that includes ***walkthroughs*** **and** ***inspections***

Code Walkthrough

 We present the code and accompanying documentation to the review team, and the team comments on their correctness.

 During walkthrough, we lead and control the discussion. The atmosphere is informal and the focus of attention is on the code, not the coder.

 Although Supervisory personnel may be present, walkthrough has no influence on the performance appraisal, consistent with the general intent of testing, finding faults, not fixing them.

Code Inspection

 Similar to Code walkthrough, but is more formal. In an inspection, review team checks the code and documentation against a prepared list of concerns.

 For eg: the team may examine the definition and use of data type and structures to see if their use is consistent with the design and with standards and procedures. The team can review algorithms and computations for their correctness and efficiency. Interfaces also checked. The team may estimate the code's performance characteristics in terms of memory usage or processing speed.

Informal Reviews

 Informal reviews include a simple desk check of a software engineering work product with a colleague, a casual meeting (involving more than two people) for the purpose of reviewing a work product

 A simple *desk check* or a *casual meeting* conducted with a colleague is a review. However, because there is no advance planning or preparation, no agenda or meeting structure, and no follow-up on the errors that are uncovered, the effectiveness of such reviews is considerably lower than more formal approaches. But a simple desk check can and does uncover errors that might otherwise propagate further into the software process.

14. Explain about software Evolution Process?

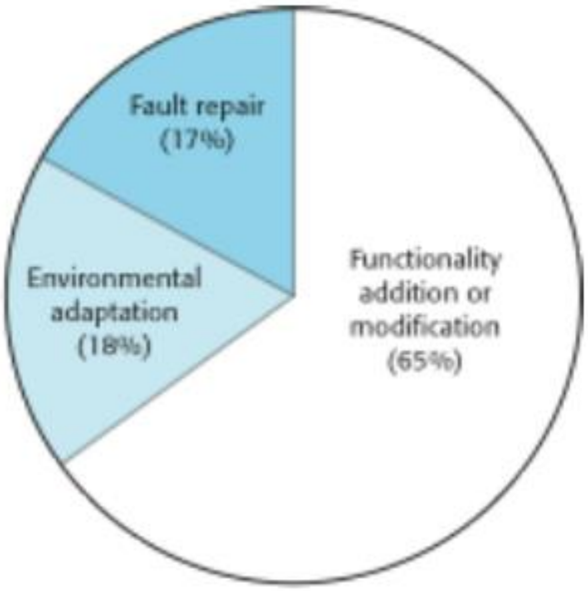
Software evolution processes depend on

- The type of software being maintained;
- The development processes used;
- The skills and experience of the people involved.
- Proposals for change are the driver for system evolution. Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- Change identification and evolution continues throughout the system lifetime.

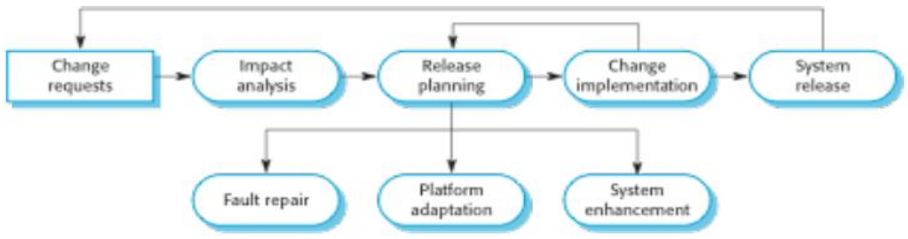
15. Write the need for software maintenance. Explain different categories of Maintenance.

Software maintenance is the process of changing, modifying, and updating software to keep up with customer needs.

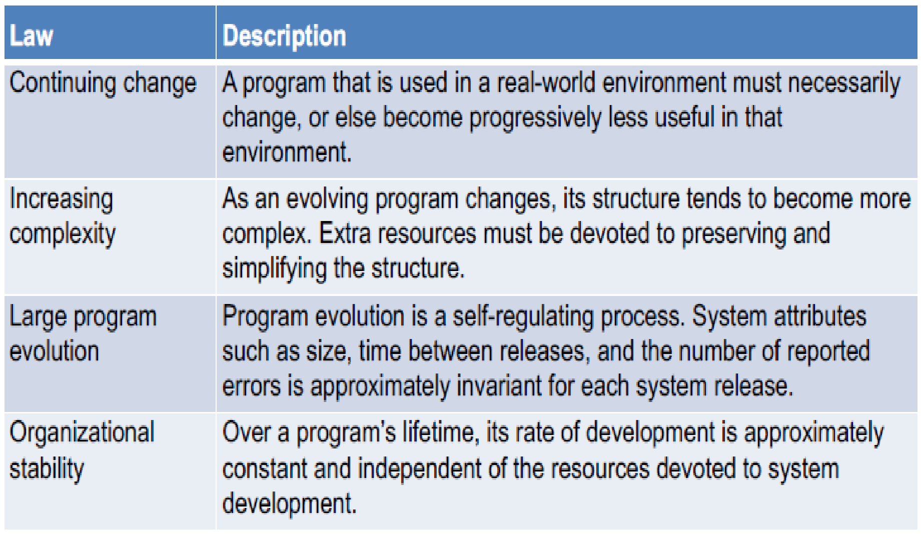
- Maintenance to repair software faults **(Corrective Maintenance)**
    - Changing a system to correct deficiencies in the way meets its requirements.
- Maintenance to adapt software to a different operating environment **(adaptive maintenance)**
    - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Maintenance to add to or modify the system's functionality **(perfective Maintenance)**
    - Modifying the system to satisfy new requirements.



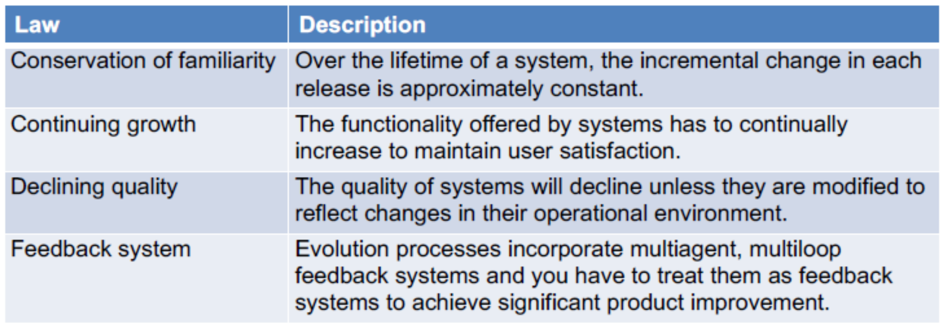
16. Explain about software evolution Process? (refer Notes)



17. State Lehman'slaws ?



Law	Description
Continuing change	A program that is used in a real-world environment must necessarily change, or else become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release.
Organizational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.



Law	Description
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will decline unless they are modified to reflect changes in their operational environment.
Feedback system	Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

18. Compare Reengineering and Refactoring

Re-engineering takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and reengineering legacy system to create a new system that is more maintainable.

Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

19. Explain about Legacy systems?

Legacy systems are old systems that are still useful and sometimes critical to business operations. They may be implemented using outdated languages and technology or may use other systems that are expensive to maintain.

20. Define Program Evolution dynamics ?

- **Program evolution dynamics** is the study of the processes of system change.
- After several major empirical studies, Lehman and Belady proposed that there were a number of 'laws' which applied to all systems as they evolved. [2]There are sensible observations rather than laws. They are applicable to large systems developed by large organizations.
- It is not clear if these are applicable to other types of software system

21. Explain about debugging ? .

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.

 Although debugging can and should be an orderly process, it is still very much an

E.1 The Debugging Process

Debugging is not testing but often occurs as a consequence of testing, the debugging process begins with the execution of a test case.

The debugging process will usually have one of two outcomes:

(1) the cause will be found and corrected or

(2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.
 Debugging has one overriding objective— to find and correct the cause of a software error or defect.

 The objective is realized by a combination of systematic evaluation, intuition, and luck. In general, three debugging strategies have been proposed: **brute force, backtracking, and**

cause elimination. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Debugging Tactics.
 The ***brute force*** **category** of debugging is probably the most common and least efficient method for isolating the cause of a software error.

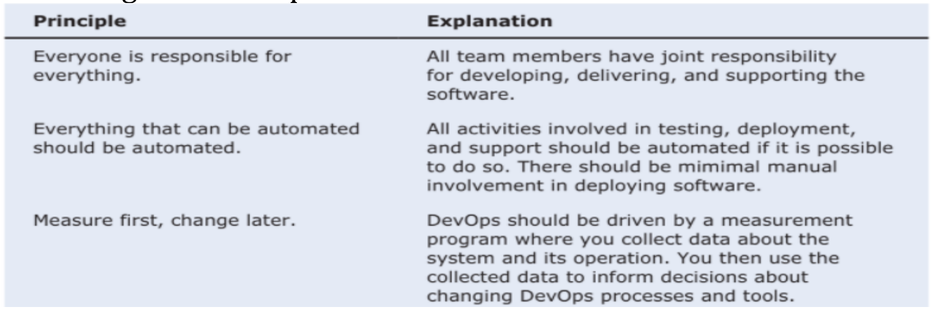
 This is the foremost common technique of debugging however is that the least economical method. during this approach, the program is loaded with print statements to print the intermediate values with the hope that a number of the written values can facilitate to spot the statement in error. This approach becomes a lot of systematic with the utilisation of a symbolic program (also known as a source code debugger), as a result of values of various variables will be simply checked and breakpoints and watch-points can be easily set to check the values of variables effortlessly.
Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

 The third approach to debugging**—** ***cause elimination***—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.

 A "cause hypothesis"is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

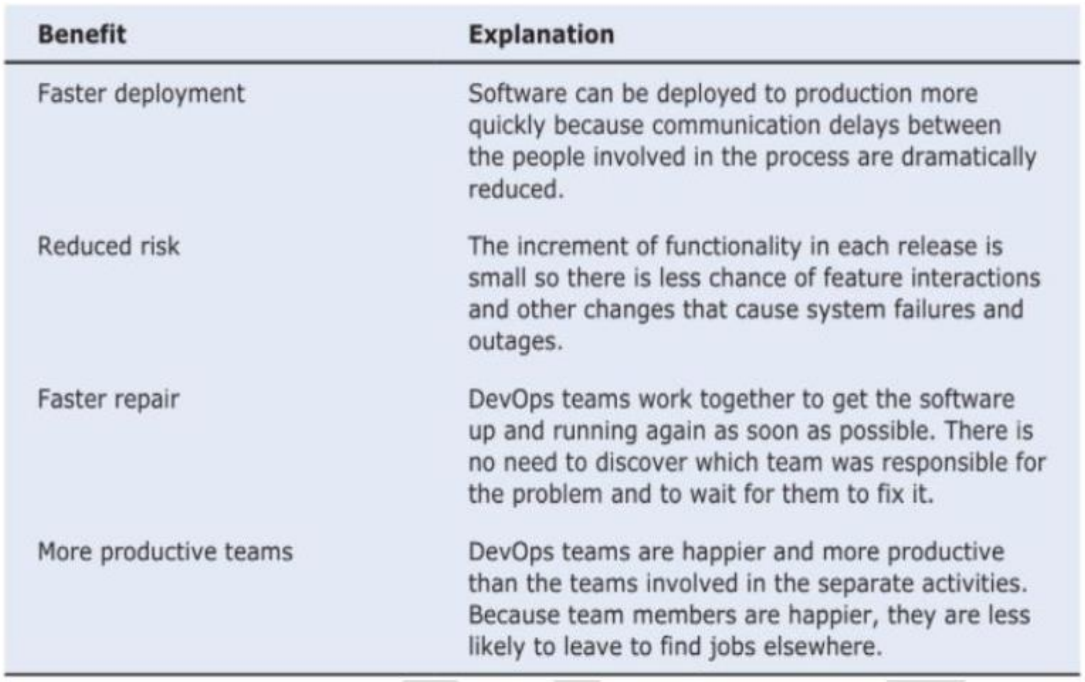
22..What is DEVOPS ? (refer notes for more explanation)

DevOps (development + operations) integrates development, deployment, and support, with a single team responsible for all software activities .



Principle	**Explanation**
Everyone is responsible for everything.	All team members have joint responsibility for developing, delivering, and supporting the software.
Everything that can be automated should be automated.	All activities involved in testing, deployment, and support should be automated if it is possible to do so. There should be mimimal manual involvement in deploying software.
Measure first, change later.	DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

23.Discuss the benefits of DEVOPS ?



Benefit	Explanation
Faster deployment	Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.
Reduced risk	The increment of functionality in each release is small so there is less chance of feature interactions and other changes that cause system failures and outages.
Faster repair	DevOps teams work together to get the software up and running again as soon as possible. There is no need to discover which team was responsible for the problem and to wait for them to fix it.
More productive teams	DevOps teams are happier and more productive than the teams involved in the separate activities. Because team members are happier, they are less likely to leave to find jobs elsewhere.

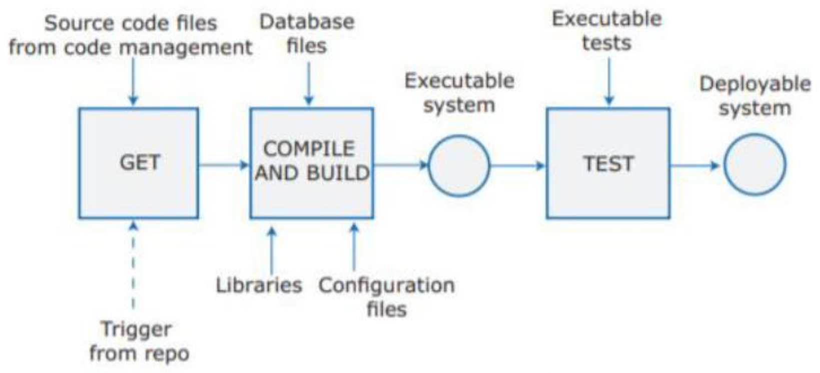
Table 2 Benefits of DevOps

24. dissuss the code management system in devops (refer notes for more explanation)

25. Explain continous integration development and DEPLOYMENT (CI/CD/CD)

System integration (system building) is the process of gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system. This involves more than compiling the system.

 Continuous integration (CI) means creating an executable version of a software system whenever a change is made to the repository. The CI tool is triggered when a file is pushed to the repo. It builds the system and runs tests on your development computer or project integration server.



- Continuous delivery means that, after making changes to a system, we ensure that the changed system is ready for delivery to customers.
- This means that you have to test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance.
- Continuous delivery does not mean that the software will necessarily be released immediately to users for deployment.

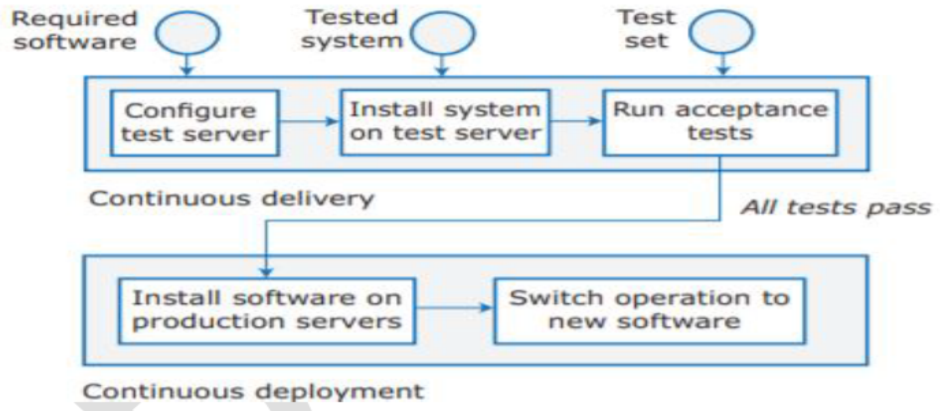


Figure 12 Continuous delivery and deployment

- Figure 12 illustrates a summarized version of this deployment pipeline, showing the stages involved in continuous delivery and deployment.
- After initial integration testing, a staged test environment is created. This is a replica of the actual production environment in which the system will run.
- The system acceptance tests, which include functionality, load, and performance tests, are then run to check that the software works as expected.

23. Explain about Test Driven development?

Test-driven development (TDD) is an approach to program development that is based on the general idea that we should write an executable test or tests for code that are writing before you write the code.

 TDD was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.

Assume that we have identified some increment of functionality to be implemented.

 Test-driven development relies on automated testing. Every time we add some functionality, we develop a new test and add it to the test suite.

 All of the tests in the test suite must pass before we move on to developing the next increment.

 Test-driven development is an approach in which executable tests are written before the code. Code is then developed to pass the tests.
The benefits of test-driven development are:
1. It is a systematic approach to testing in which tests are clearly linked to
section of the program code.
2. The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests..
3. Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system

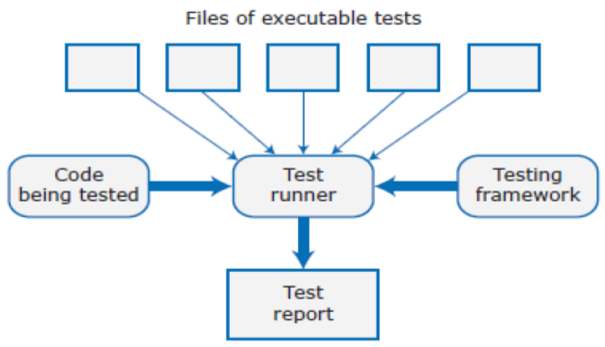
24. Explain about Test documentation ?

Documentation testing can be approached in two phases.

- The first phase, technical review examines the document for editorial clarity.
- The second phase, live test, uses the documentation in conjunction with the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed earlier.
Graph-based testing can be used to describe the use of the program;
equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions.
MBT can be used to ensure that documented behavior and actual behavior coincide.
Program usage is then tracked through the documentation.

24. Explain about Test Automation?



- Automated testing (Figure 9.4) is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result, and a check that the unit returns the expected result.
- we run the test and the test passes if the unit returns the expected result. Normally, we should develop hundreds or thousands of executable tests for a software product.
- The development of automated testing frameworks, such as JUnit for Java in the 1990s, reduced the effort involved in developing executable tests.
- Testing frameworks are now available for all widely used programming languages. A suite of hundreds of unit tests, developed using a framework, can be run on a desktop computer in a few seconds. A test report shows the tests that have passed and failed.
- Testing frameworks provide a base class, called something like "TestCase" that is used by the testing framework. To create an automated test, you define your own test class as a subclass of this TestCase class. Testing frameworks include a way of running all of the tests defined in the classes that are based on TestCase and reporting the results of the tests.

It is good practice to structure automated tests in three parts:

1. *Arrange* You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.

2. *Action* You call the unit that is being tested with the test parameters.

3. *Assert* You make an assertion about what should hold if the unit being tested has executed successfully.