



SOFTWARE CODING AND TESTING

Prepared by:
Prof. Sherin Mariam Jijo

CONTENTS

- ❖ **Coding Standard and coding Guidelines**
- ❖ **Code Review**
- ❖ **Software Documentation**
- ❖ **Testing Strategies**
- ❖ **Testing Techniques**
- ❖ **Test Suites Design**
- ❖ **Testing Conventional Applications**
- ❖ **Testing Object Oriented Applications**
- ❖ **Testing Web and Mobile Applications**
- ❖ **Testing Tools (Win runner, Load runner).**

CODING

- Good software development organizations normally require their programmers to adhere to some well- defined and standard style of coding called coding standards.
- The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:
 - A coding standard gives a uniform appearance to the codes written by different engineers.
 - It enhances code understanding.
 - It encourages good programming practices.
- A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

CODING STANDARDS AND GUIDELINES

❑ Rules for limiting the use of global:

- These rules list what types of data can be declared global and what cannot.

❑ Contents of the headers preceding codes for different modules:

- The information contained in the headers of different modules should be standard for an organization.
- The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:
 - Name of the module.
 - Date on which the module was created.
 - Modification history.
 - Different functions supported, along with their input/output parameters.

Global variables accessed/modified by the module

❑ **Naming conventions for global variables, local variables, and constant identifiers:**

- A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.

❑ **Error return conventions and exception handling mechanisms:**

- The way error conditions are reported by different functions in a program are handled should be standard within an organization.

❑ **Do not use a coding style that is too clever or too difficult to understand:**

- Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can have an ambiguous meaning of the code and delay understanding.

❑ **Avoid obscure side effects:**

- The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations.
- An unclear side effect is one that is not obvious from a casual examination of the code.

❑ **Do not use an identifier for multiple purposes:**

- Programmers often use the same identifier to denote several temporary entities.
- For example, some programmers use a temporary loop variable for computing and a storing the final result.
- Each variable should be given a descriptive name indicating its purpose.

☐ **The code should be well-documented:**

- As a rule of thumb, there must be at least one comment line on the average for every three-source line.

☐ **The length of any function should not exceed 10 source lines:**

- A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions.

☐ **Do not use goto statements:**

- Use of goto statements makes a program unstructured and makes it very difficult to understand.

CODE REVIEW


- ❖ Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated.
- ❖ Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module.
- ❖ There are two types' code review techniques are **code inspection and code walk through.**

CODE WALK THROUGH

- ❖ Code walk through is an informal code analysis technique.
- ❖ In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated.
- ❖ A few members of the development team are given the code few days before the walk through meeting to read and understand code.
- ❖ Each member selects some test cases and simulates execution of the code by hand.
- ❖ The main objectives of the walk through are to discover the algorithmic and logical errors in the code.
- ❖ Even though a code walk through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective.

CODE INSPECTION

- ❖ In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.
- ❖ In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk through.
- ❖ For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter.

- 
- ❖ It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure.
 - ❖ In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.
 - ❖ Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed.

SOFTWARE DOCUMENTATION

- ❖ When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc. are also developed as part of any software engineering process.
- ❖ All these documents are a vital part of good software development practice.
- ❖ Different types of software documents can broadly be classified into the following:
 - **Internal documentation**
 - **External documentation**

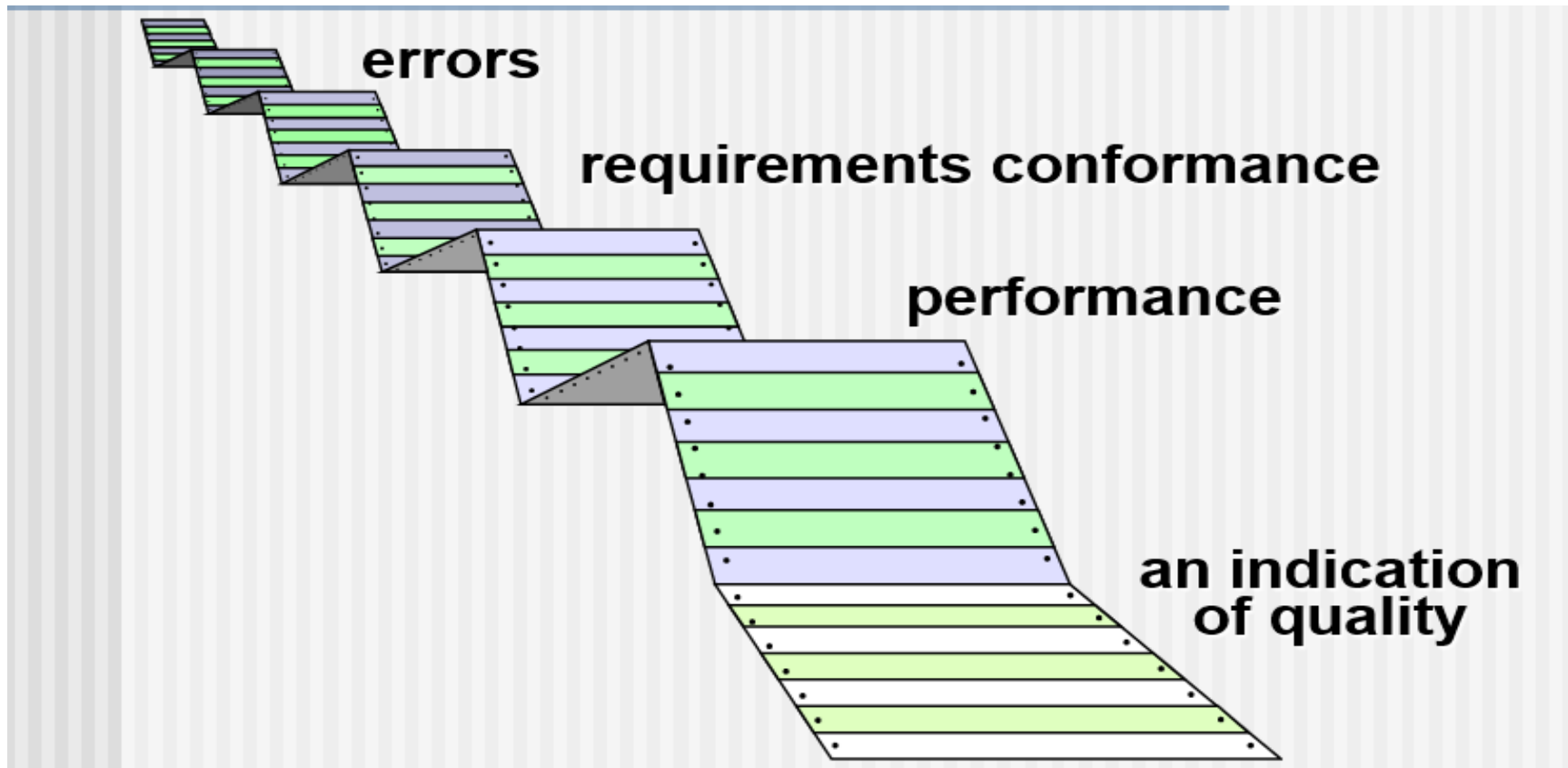
- ❖ **Internal documentation** is the code comprehension features provided as part of the source code itself.
- ❖ Internal documentation is provided through appropriate module headers and comments embedded in the source code.
- ❖ Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc.
- ❖ This is of course in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without thought.

- ❖ But even when code is carefully commented, meaningful variable names still are more helpful in understanding a piece of code.
- ❖ Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines.
- ❖ **External documentation** is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc.
- ❖ A systematic software development style ensures that all these documents are produced in an orderly fashion.

SOFTWARE TESTING

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

WHAT TESTING SHOWS



TESTING CHARACTERISTICS

- ❖ To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- ❖ Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- ❖ Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- ❖ Testing is conducted by the developer of the software and (for large projects) an independent test group.
- ❖ Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

V & V

- ❖ *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
- ❖ *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:
 - ❖ *Verification*: "Are we building the product right?"
 - ❖ *Validation*: "Are we building the right product?"

Verification

- It includes checking documents, design, codes and programs.
- Verification is the static testing.
- It does *not* include the execution of the code.
- Methods used in verification are reviews, walkthroughs, inspections and desk-checking.
- It checks whether the software conforms to specifications or not.
- It can find the bugs in the early stage of the development.
- The goal of verification is application and software architecture and specification.
- Quality assurance team does verification.
- It comes before validation.
- It consists of checking of documents/files and is performed by human.

Validation

- It includes testing and validating the actual product.
- Validation is the dynamic testing.
- It includes the execution of the code.
- Methods used in validation are Black Box Testing, White Box Testing and non-functional testing.
- It checks whether the software meets the requirements and expectations of a customer or not.
- It can only find the bugs that could not be found by the verification process.
- The goal of validation is an actual product.
- Validation is executed on software code with the help of testing team.
- It comes after verification.
- It consists of execution of program and is performed by computer.

Who tests the Software?



developer

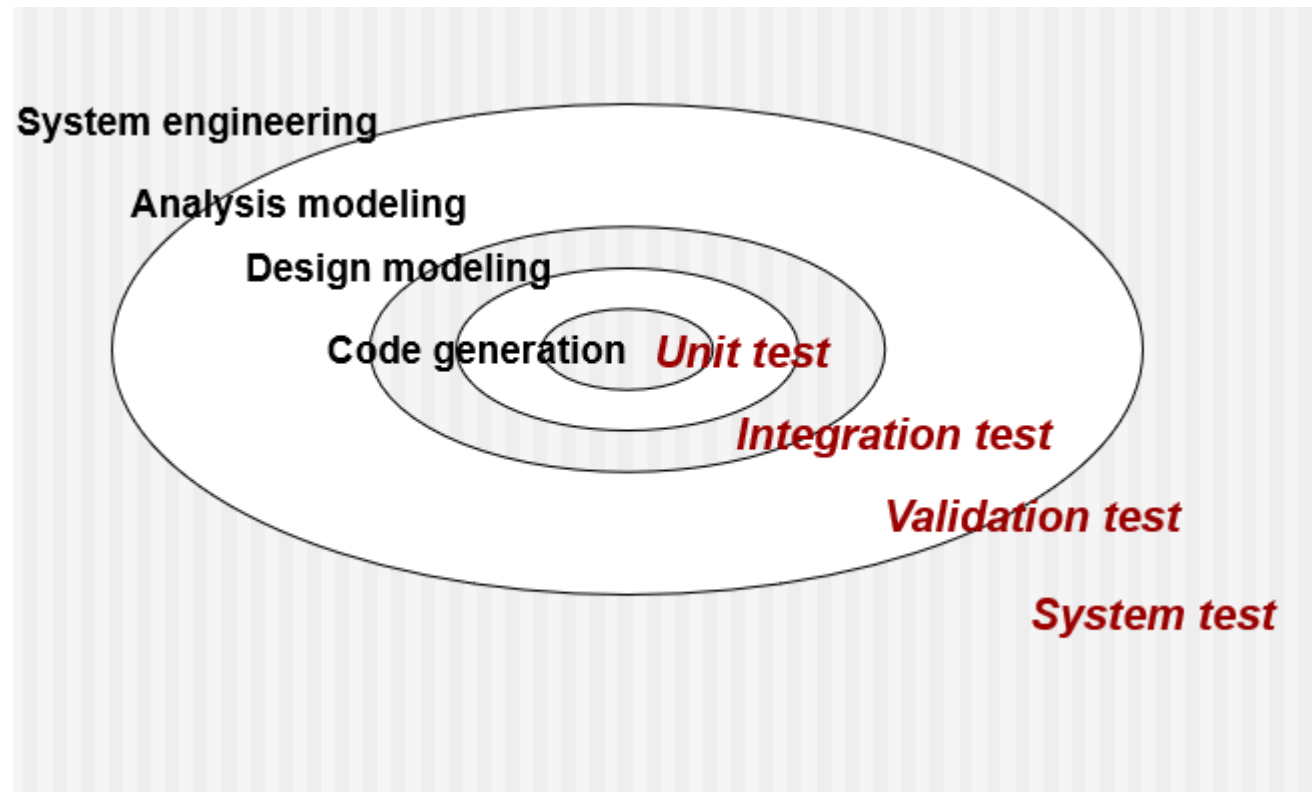
Understands the system
but, will test "gently"
and, is driven by "delivery"




independent tester

Must learn about the system,
but, will attempt to break it
and, is driven by quality

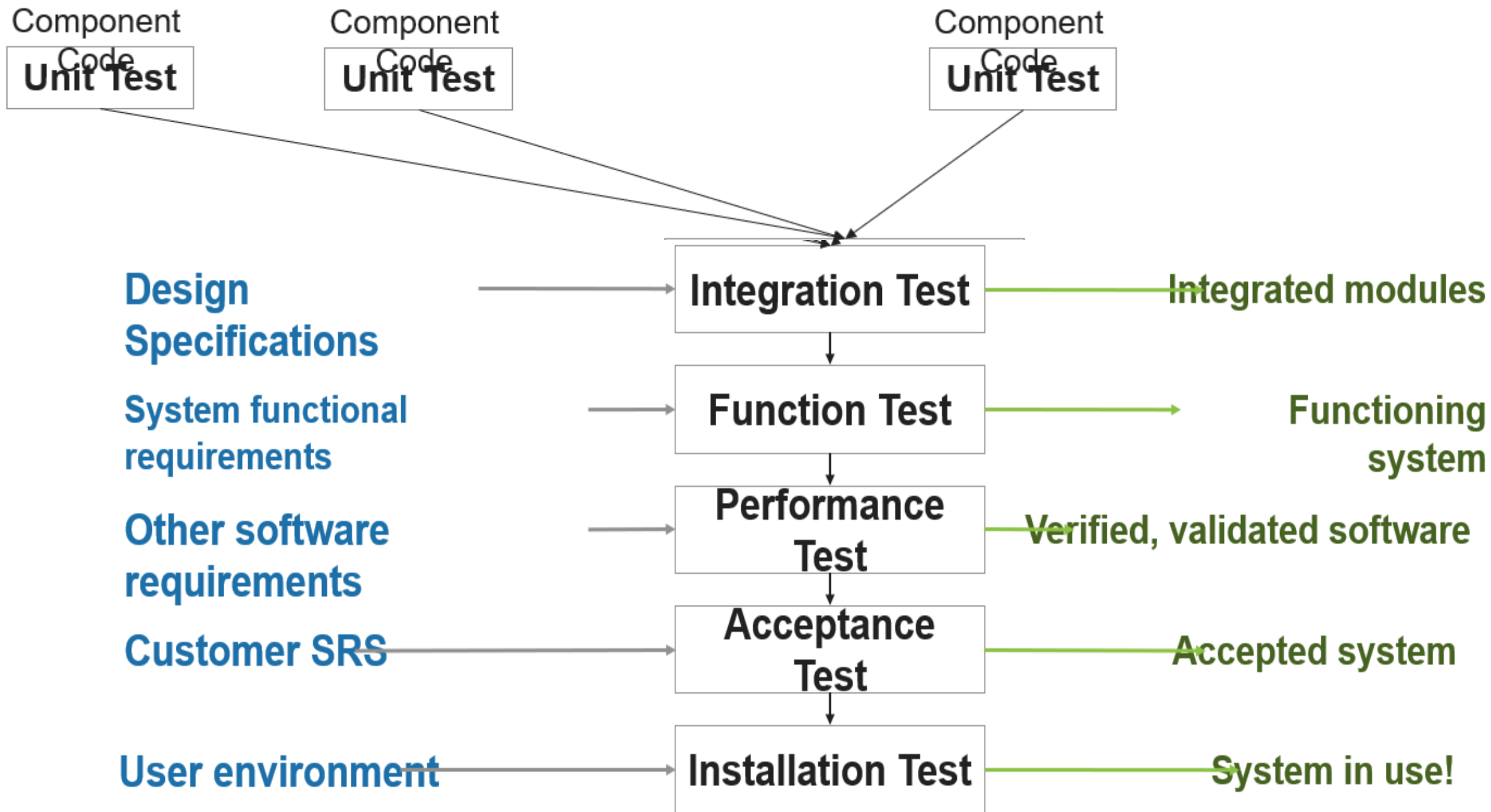
SOFTWARE TESTING STRATEGY



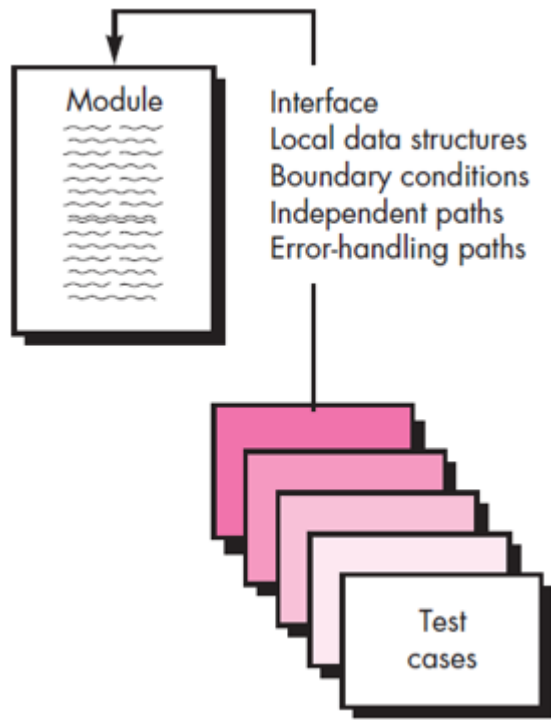
- 
- Initially, system engineering defines a role of software and leads to sw requirements analysis.
 - Here the information domain, function, behavior, performance, constraints and validation criteria for sw are established.
 - Moving inward along the spiral, you come to design and finally to coding.
 - We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’.

STRATEGIC ISSUES

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself.
- Use effective technical reviews as a filter prior to testing.
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

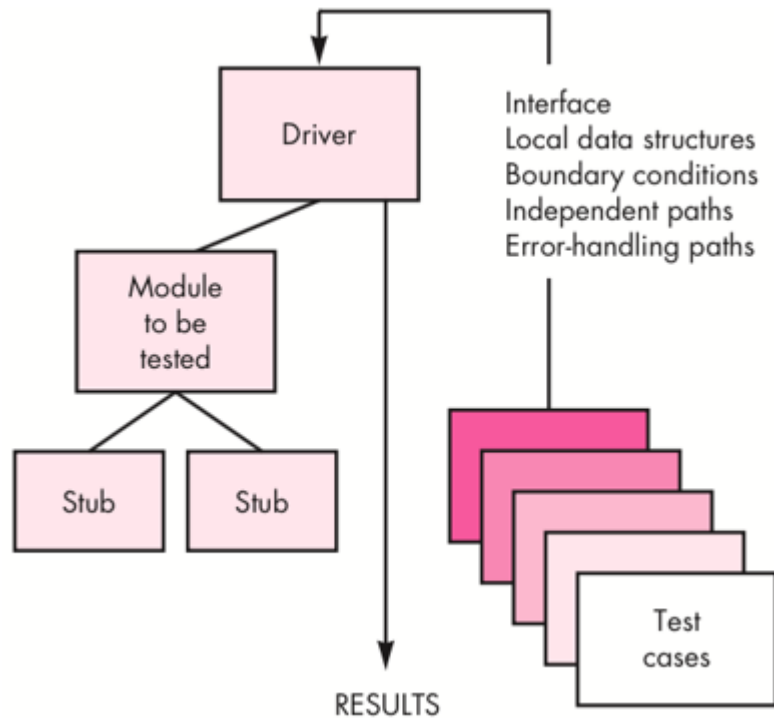


UNIT TESTING:



- Unit is the smallest part of a software system which is testable it may include code files, classes and methods which can be tested individual for correctness.

- Unit is a process of validating such small building block of a complex system, much before testing an integrated large module or the system as a whole.



- Driver and/or stub software must be developed for each unit test a driver is nothing more than a "main program" that accepts test case data, passes such data to the component, and prints relevant results.
- Stubs serve to replace modules that are subordinate (called by) the component to be tested.
- A stub or "dummy subprogram" uses the subordinate module's interface.

INTEGRATION TESTING

- ❑ Integration is defined as a set of integration among component.
- ❑ Testing the interactions between the module and interactions with other system externally is called Integration Testing.
- ❑ Testing of integrated modules to verify combined functionality after integration.
- ❑ Integration testing addresses the issues associated with the dual problems of verification and program construction.
- ❑ Modules are typically code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.

- Tendency to attempt non incremental integration, ie. To construct the program using a “big bang” approach.
- All components are combined in advance.
- The entire program is tested as a whole.
- And chaos usually results.

Solution

- Incremental integration
- Errors are easy to isolate and correct.
- Interfaces are more likely to be tested completely.



- ❑ Top down testing

- ❑ Bottom-up testing

- ❑ Sandwich testing

Top down testing

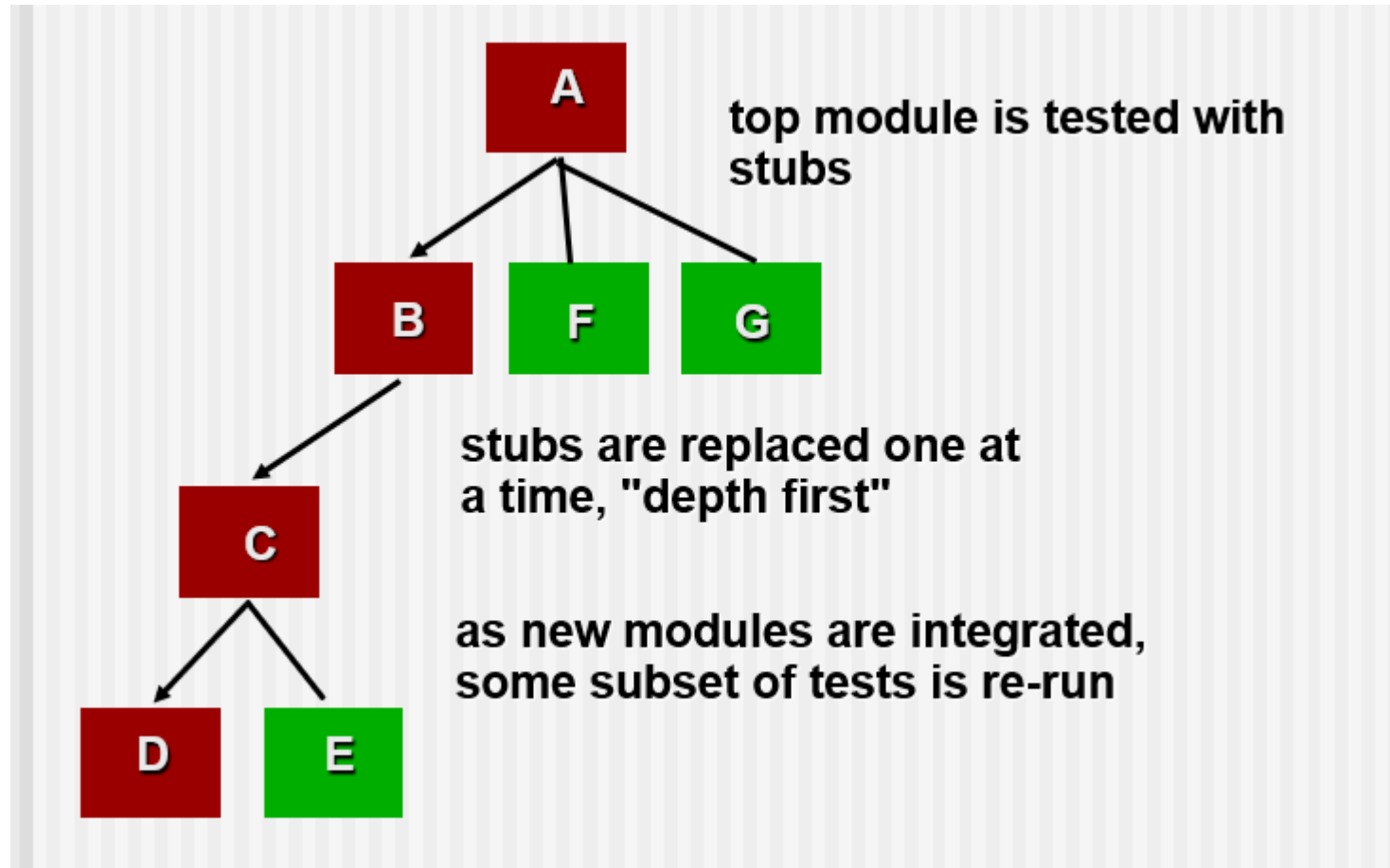
Top-down testing is a type of incremental integration testing approach in which testing is done by integrating or joining two or more modules by moving down from top to bottom through control flow of architecture structure.

In these, high-level modules are tested first, and then low-level modules are tested.

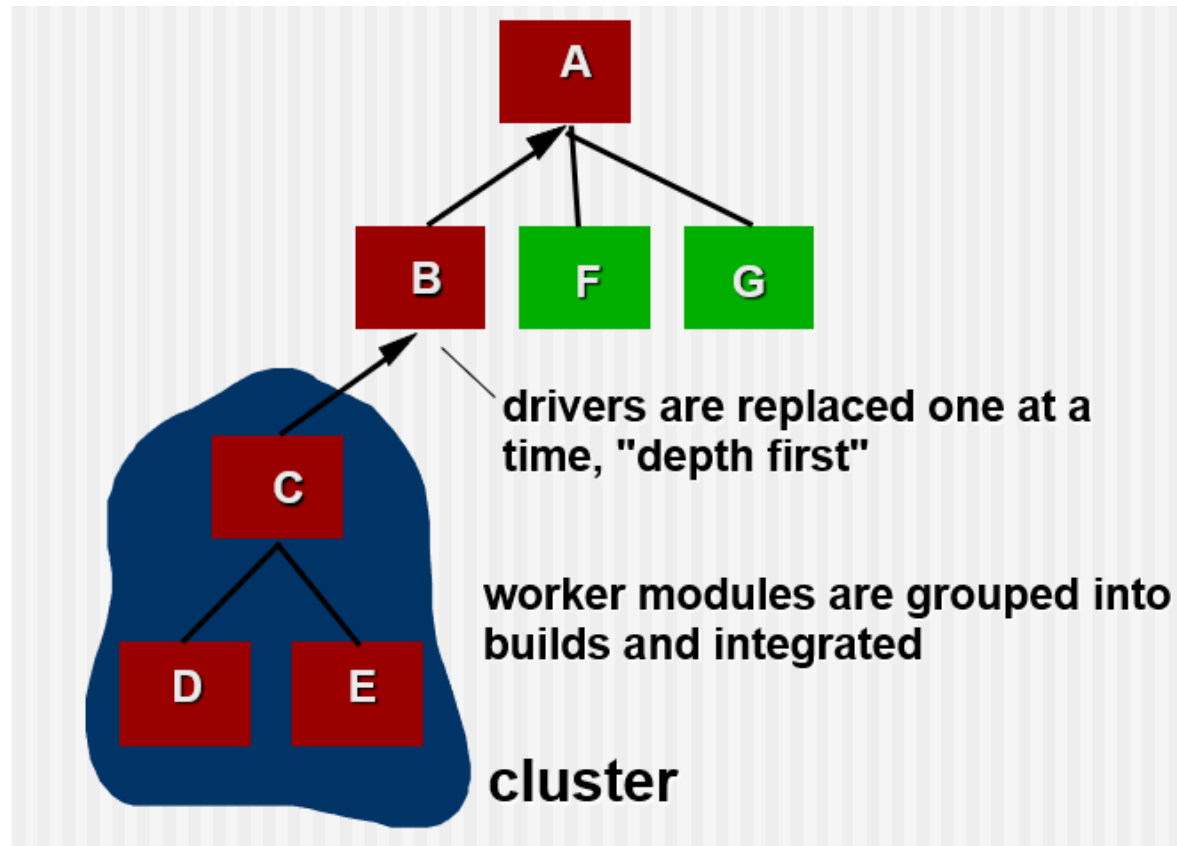
Then, finally, integration is done to ensure that system is working properly. Stubs and drivers are used to carry out this project.

This technique is used to increase or stimulate behavior of Modules that are not integrated into a lower level.

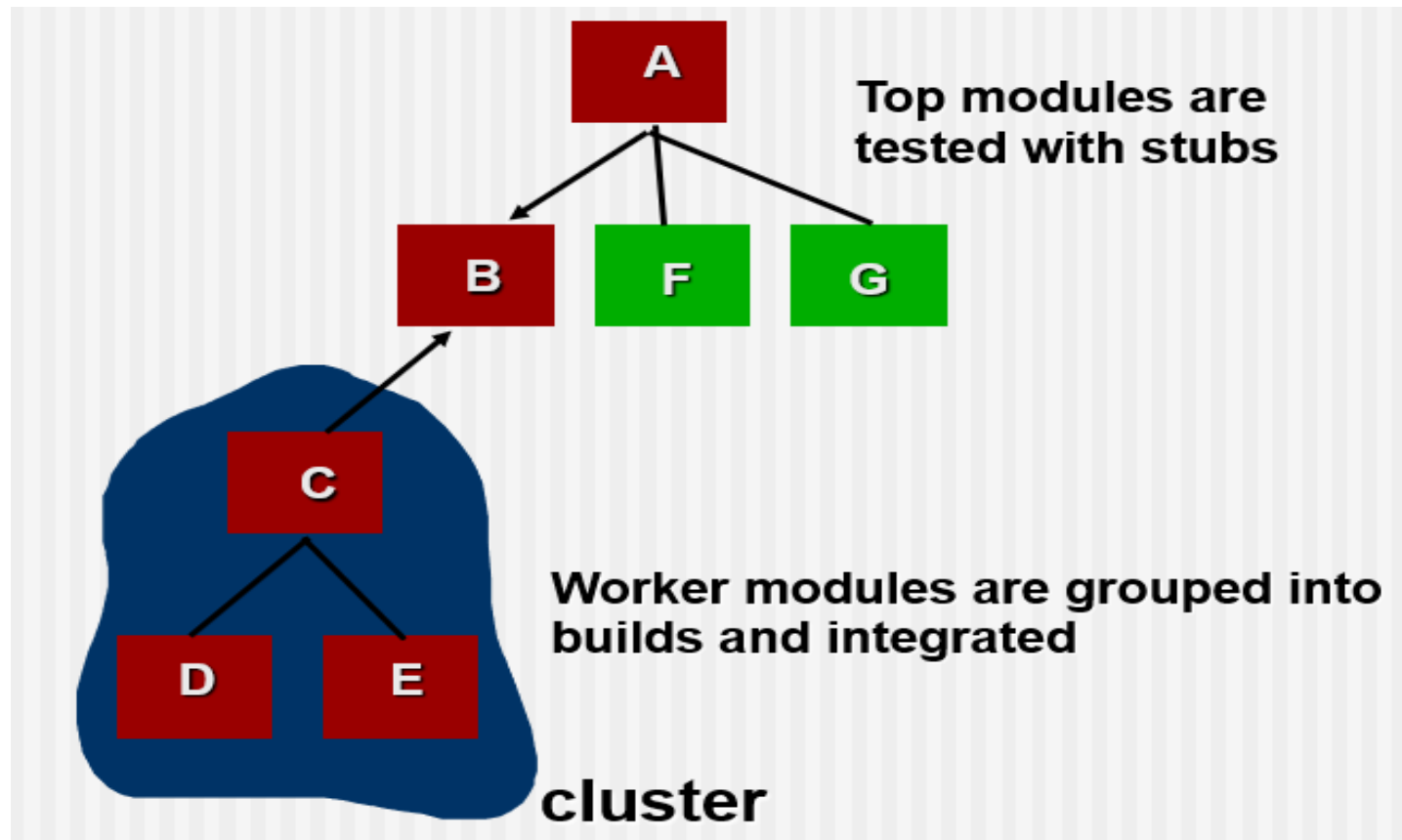
Top down approach



Bottom up approach



Sandwich testing



VALIDATION TESTING

- ❑ The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.
- ❑ Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.
- ❑ Validation testing provides final assurance that software meets all informational, functional, behavioral, and performance requirements.

Alpha test

- The alpha test is conducted at the developer's site by a representative group of end users.
- The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.
- Alpha tests are conducted in a controlled environment.

Beta test

- The beta test is conducted at one or more end-user sites.
- Unlike alpha testing, the developer generally is not present.
- Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer.

Regression testing

- *Regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

SMOKE TESTING

A common approach for creating “daily builds” for product software

Smoke testing steps:

- Software components that have been translated into code are integrated into a “build.”
- A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
- A series of tests is designed to expose errors that will keep the build from properly performing its function.
- The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
- The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
- The integration approach may be top down or bottom up.

SYSTEM TESTING

- In system testing the software and other system elements are tested as a whole.
- To test computer software, you spiral out in a clockwise direction along streamlines that increase the scope of testing with each turn.
- System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

Recovery Testing

Security Testing

Stress Testing

Performance Testing

Deployment Testing

- **Recovery testing** is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

- If recovery is automatic (performed by the system itself), re initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness.

If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

- **Security testing** attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

- During security testing, the tester plays the role(s) of the individual who desires to penetrate the system.

- **Stress testing** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
 - A variation of stress testing is a technique called sensitivity testing.
 - Performance testing is designed to test the run-time performance of software within the context of an integrated system.
-
- **Performance testing** occurs throughout all steps in the testing process.
 - Even at the unit level, the performance of an individual module may be assessed as tests are conducted.

- **Deployment testing**, sometimes called configuration testing, exercises the software in each environment in which it is to operate.
- In addition, deployment testing examines all installation procedures and specialized installation software that will be used by customers, and all documentation that will be used to introduce the software to end users.

ACCEPTANCE TESTING

- Acceptance Testing is a level of the software testing where a system is tested for acceptability.
- The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.
- It is a formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.
- Acceptance Testing is performed after System Testing and before making the system available for actual use.

WHITE BOX TESTING

- White-box testing, sometimes called glass-box testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.
- Using white-box testing methods, you can derive test cases that
 1. Guarantee that all independent paths within a module have been exercised at least once.
 2. Exercise all logical decisions on their true and false sides.
 3. Execute all loops at their boundaries and within their operational bounds.
 4. Exercise internal data structures to ensure their validity.

- White Box Testing method is applicable to the following levels of software testing:
 - It is mainly applied to Unit testing and Integration testing
 - Unit Testing: For testing paths within a unit.
 - Integration Testing: For testing paths between units.
 - System Testing: For testing paths between subsystems

White box testing advantages


- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
- Testing is more thorough, with the possibility of covering most paths.

White box testing disadvantages

- Since tests can be very complex, highly skilled resources are required, with thorough knowledge of programming and implementation.
- Test script maintenance can be a burden if the implementation changes too frequently.
- Since this method of testing is closely tied with the application being testing, tools to cater to every kind of implementation/platform may not be readily available.

BLACK BOX TESTING

- Black-box testing, also called behavioral 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 is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white box methods.

- 
- 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
 - 5 Initialization and termination errors.

- Black Box Testing method is applicable to the following levels of software testing:
 - It is mainly applied to System testing and Acceptance testing
 - Integration Testing
 - System Testing
 - Acceptance Testing
- The higher the level, and hence the bigger and more complex the box, the more black box testing method comes into use.

Black box testing advantages

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- Tester need not know programming languages or how the software has been implemented.
- Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
- Test cases can be designed as soon as the specifications are complete.

Black box testing disadvantages

- Only a small number of possible inputs can be tested and many program paths will be left untested.
- Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
- Tests can be redundant if the software designer/ developer has already run a test case.

QUALITY FUNCTION DEPLOYMENT (QFD)

- Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
- QFD “concentrates on maximizing customer satisfaction from the software engineering process”
- To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.
- QFD identifies three types of requirements:

1. Normal requirements

- The objectives and goals that are stated for a product or system during meetings with the customer.
- If these requirements are present, the customer is satisfied.

Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

2. Expected requirements

- These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them.
- Their absence will be a cause for significant dissatisfaction.

Exciting requirements

- These features go beyond the customer's expectations and prove to be very satisfying when present.
- For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities that delight every user of the product.


Although QFD concepts can be applied across the entire software process, specific QFD techniques are applicable to the requirements elicitation activity.

QFD uses customer interviews and observation, surveys, and examination of historical data as raw data for the requirements gathering activity.

TESTING CONVENTIONAL APPLICATIONS

Software is tested from two different perspectives:

1. Internal program logic is exercised using white box test case design techniques
2. Software requirements are exercised using black box testcase design techniques.

- 
- Use cases assist in the design of tests to uncover errors at the software validation level.
 - In every case the intent is to find the maximum number of errors with the minimum amount of effort and time.
 - A set of test cases designed to exercise both internal logic, interface, component collaborations, and external requirements is designed and documented, expected results are defined, and actual results are recorded.

Software Testing Fundamentals

The following characteristics lead to testable software.

Operability.

- “The better it works, the more efficiently it can be tested.”
- If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

Observability.

- “What you see is what you test.”
- Inputs provided as part of testing produce distinct outputs.
- System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

Controllability.

- “The better we can control the software, the more the testing can be automated and optimized.”
- All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured.
- All code is executable through some combination of input. Software and hardware states and variables can be controlled directly by the test engineer.
- Tests can be conveniently specified, automated, and reproduced.

Decomposability.

- “By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.”
- The software system is built from independent modules that can be tested independently.

Simplicity.

- “The less there is to test, the more quickly we can test it.”
- The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to meet requirements); structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability.

- “The fewer the changes, the fewer the disruptions to testing.”
- Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests.
- The software recovers well from failures.

Understandability.

- “The more information we have, the smarter we will test.”
- The architectural design and the dependencies between internal, external, and shared components are well understood.
- Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

CYCLOMATIC COMPLEXITY

Cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it.

It is a software metric used to indicate the complexity of a program.

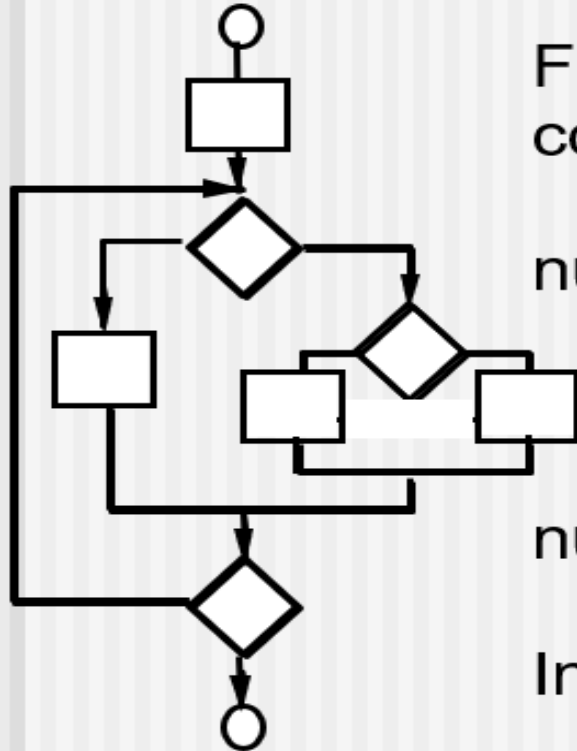
It is computed using the Control Flow Graph of the program.

The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if second command might immediately follow the first command.

Studies indicate that the higher $V(G)$, the higher the probability of errors.

CYCLOMATIC COMPLEXITY

Basis Path Testing



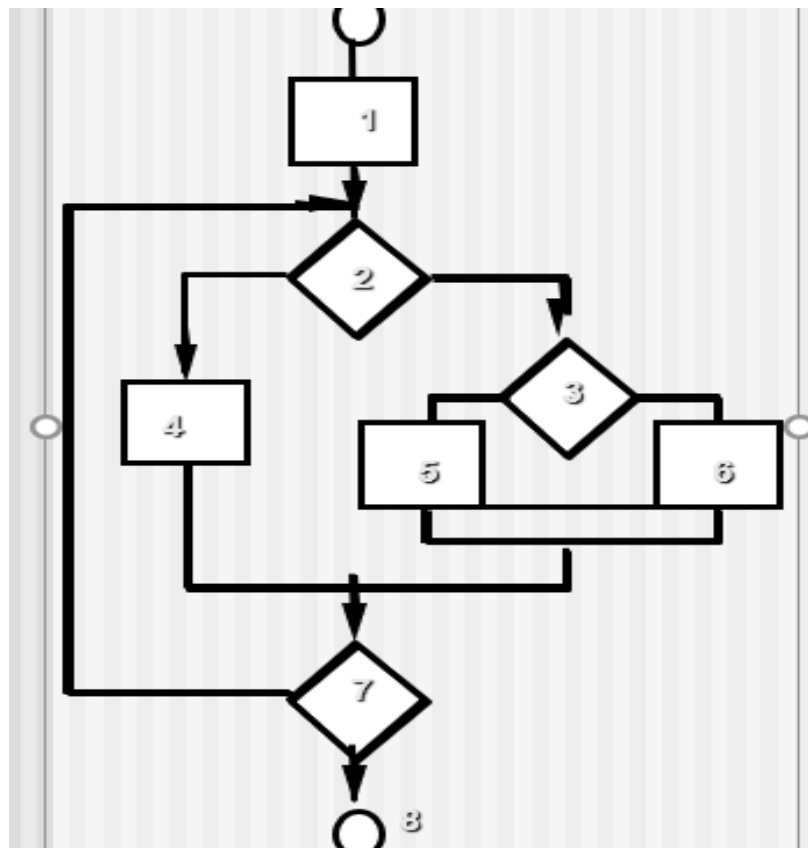
First, we compute the cyclomatic complexity:

number of simple decisions + 1

or

number of enclosed areas + 1

In this case, $V(G) = 4$



Next, we derive the independent paths:

Since $V(G) = 4$, there are four paths

Path 1: 1,2,3,6,7,8

Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

USE OF CYCLOMATIC COMPLEXITY:

Determining the independent path executions thus proven to be very helpful for Developers and Testers.

It can make sure that every path have been tested at least once.

Thus help to focus more on uncovered paths.

Code coverage can be improved.


Risk associated with program can be evaluated.

These metrics being used earlier in the program helps in reducing the risks.

TESTING OBJECT ORIENTED APPLICATIONS

Unit Testing in the OO Context

- When object-oriented software is considered, the concept of the unit changes.
- Encapsulation drives the definition of classes and objects.
- This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as methods or services) that manipulate these data.
- Rather than testing an individual module, the smallest testable unit is the encapsulated class.

- 
- Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically.
 - Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flows across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

INTEGRATION TESTING IN THE OO CONTEXT

- Because object-oriented software does not have a hierarchical control structure, conventional top-down and bottom-up integration strategies have little meaning.
- In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the “direct and indirect interactions of the components that make up the class”.
- There are two different strategies for integration testing of OO systems.
- The first, **thread-based testing**, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually.

- **Regression testing** is applied to ensure that no side effects occur.
- The second integration approach, **use-based testing**, begins the construction of the system by testing those classes (called independent classes) that use very few (if any) of server classes.
- After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested.
- **Cluster testing** is one step in the integration testing of OO software.
- Here, a cluster of collaborating classes (determined by examining the CRC and object relationship model) is exercised by designing test cases that attempt to uncover.

Validation Testing in an OO Context

- At the validation or system level, the details of class connections disappear.
- Like conventional validation, the validation of OO software focuses on user-visible actions and user-recognizable outputs from the system.
- To assist in the derivation of validation tests, the tester should draw upon use cases that are part of the requirements model.
- The use case provides a scenario that has a high likelihood of uncovered errors in user-interaction requirements.
- Conventional black-box testing methods can be used to drive validation tests.

TESTING WEB APPLICATIONS

- WebApp testing is a collection of related activities with a single goal: to uncover errors in WebApp content, function, usability, navigability, performance, capacity, and security.
- To accomplish this, a testing strategy that encompasses both reviews and executable testing is applied.

STEPS

❑ The WebApp testing process begins by focusing on user visible aspects of the WebApp and proceeds to tests that exercise technology and infrastructure.

❑ Seven testing steps are performed.

1. Content testing
2. Interface testing
3. Navigation testing
4. Component testing
5. Configuration testing
6. Performance testing
7. Security testing

DIMENSIONS OF QUALITY

Content is evaluated at both a syntactic and semantic level. At the syntactic level, spelling, punctuation, and grammar are assessed for text-based documents. At a semantic level, correctness (of information presented), Consistency (across the entire content object and related objects), and lack of ambiguity are all assessed.

Function is tested to uncover errors that indicate lack of conformance to customer requirements. Each WebApp function is assessed for correctness, instability, and general conformance to appropriate implementation standards (e.g., Java or AJAX language standards).

Structure is assessed to ensure that it properly delivers WebApp content and unction, that it is extensible, and that it can be supported as new content or functionality is added.

Usability is tested to ensure that each category of user is supported by the interface and can learn and apply all required navigation syntax and semantics.

Navigability is tested to ensure that all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, and erroneous links).

Performance is tested under a variety of operating conditions, configurations, and loading to ensure that the system is responsive to user interaction and handles extreme loading without unacceptable operational degradation.

Compatibility is tested by executing the WebApp in a variety of different host configurations on both the client and server sides. The intent is to find errors that are specific to a unique host configuration.

Interoperability is tested to ensure that the WebApp properly interfaces with other applications and/or databases.

Security is tested by assessing potential vulnerabilities and attempting to exploit each. Any successful penetration attempt is deemed a security failure.

Content Testing

- Errors in WebApp content can be as trivial as minor typographical errors or as significant as incorrect information, improper organization, or violation of intellectual property laws.
- Content testing attempts to uncover these and many other problems before the user encounters them.
- Content testing combines both reviews and the generation of executable test cases.
- Reviews are applied to uncover semantic errors in content.
- Executable testing is used to uncover content errors that can be traced to dynamically derived content that is driven by data acquired from one or more databases.

User Interface Testing

- Verification and validation of a WebApp user interface occurs at three distinct points.
- During requirements analysis, the interface model is reviewed to ensure that it conforms to stakeholder requirements and to other elements of the requirements model.
- During design the interface design model is reviewed to ensure that generic quality criteria established for all user interfaces have been achieved and that application-specific interface design issues have been properly addressed.
- During testing, the focus shifts to the execution of application-specific aspects of user interaction as they are manifested by interface syntax and semantics.
- In addition, testing provides a final assessment of usability.

Component-Level Testing

- Component-level testing, also called function testing, focuses on a set of tests that attempt to uncover errors in WebApp functions.
- Each WebApp function is a software component (implemented in one of a variety of programming or scripting languages) and can be tested using black-box (and in some cases, white-box) techniques.
- Component-level test cases are often driven by forms-level input. Once forms data are defined, the user selects a button or other control mechanism to initiate execution.

Navigation Testing

- The job of navigation testing is to ensure that the mechanisms that allow the WebApp user to travel through the WebApp are all functional and to validate that each navigation semantic unit (NSU) can be achieved by the appropriate user category.
- Navigation mechanisms should be tested are Navigation links, Redirects, Bookmarks, Frames and framesets, Site maps, Internal search engines.

Configuration Testing

- Configuration variability and instability are important factors that make WebApp testing a challenge. Hardware, operating system(s), browsers, storage capacity, network communication speeds, and a variety of other client-side factors are difficult to predict for each user.
- One user's impression of the WebApp and the manner in which she interacts with it can differ significantly from another user's experience, if both users are not working within the same client-side configuration.
- The job of configuration testing is not to exercise every possible client-side configuration.
- Rather, it is to test a set of probable client-side and server-side configurations to ensure that the user experience will be the same on all of them and to isolate errors that may be specific to a particular configuration.

Security Testing

- Security tests are designed to probe vulnerabilities of the client-side environment, the network communications that occur as data are passed from client to server and back again, and the server-side environment.
- Each of these domains can be attacked, and it is the job of the security tester to uncover weaknesses that can be exploited by those with the intent to do so.

Performance Testing

- Performance testing is used to uncover performance problems that can result from lack of server-side resources, inappropriate network bandwidth, inadequate database capabilities, faulty or weak operating system capabilities, poorly designed WebApp functionality, and other hardware or software issues that can lead to degraded client-server performance.

Consider the program given below

```
void main()
{
int i,j,k;
readln (i,j,k);
if( (i < j) || ( i > k) )
{
writeln("then part");
if (j < k)
writeln ("j less then k");
else writeln ( " j not less then k");
}
else writeln( "else Part"); }
```

- (i) Draw the flow graph.
- (ii) Determine the cyclomatic complexity.
- (iii) Arrive at all the independent paths.