CSE1005: Software Engineering

Module No. 3: Software Testing

Dr. K Srinivasa Reddy
SCOPE, VIT-AP University

Module No. 3: Software Testing

Strategic Approach to Software Testing, Strategic Issues, Test Strategies for Conventional Software, Fundamentals, Black box Testing, White box testing.

Text Book:

1. Roger Pressman, “Software Engineering: A Practitioner’s Approach”, McGraw-Hill, 7th Edition, 2016.

Course Outcome: Apply and check for errors in software using software testing techniques (CO3)

Transferring the amount to friend's account by using Net Banking / UPI

- Initiate the transaction, get a successful transaction message, and the amount also deducted from your account.
- However, your friend confirms that they didn’t received any amount yet.
- Likewise, your account is also not reflecting the reversed transaction.
- This will surely make you upset and leave you as an unsatisfied customer.
- **Why did it happen?**
  - **Improper testing** of the net banking / UPI application before the release.
  - Thorough testing of the application for all possible user operations would lead to early identification of this type of problems.
  - Fix the errors before releasing it to the public for a smoother experience.

Software Testing – Introduction

Sources of Problems

- **Requirements Definition:** Erroneous, incomplete, inconsistent requirements.
- **Design:** Fundamental design flaws in the software.
- **Implementation:** Mistakes in writing, programming faults, malicious code, etc.
- **Support Systems:** Poor programming languages, faulty compilers and debuggers, misleading development tools.
- **Inadequate Testing of Software:** Incomplete testing, poor verification, mistakes in debugging.
- **Evolution:** Sloppy redevelopment or maintenance, introduction of new flaws in attempts to fix old flaws, incremental escalation to inordinate complexity.

Software Testing – Introduction

Adverse Effects of Faulty Software in various domains:

- **Communications:** Loss or corruption of communication media, non delivery of data.
- **Space Applications:** Lost lives, launch delays.
- **Defense and Warfare:** Misidentification of targets
- **Transportation:** Deaths, delays, sudden acceleration, fire, inability to brake, .
- **Safety-critical Applications:** Death, injuries.
- **Electric Power:** Death, injuries, power outages, long-term health hazards
- **Money Management:** Fraud, violation of privacy, shutdown of stock exchanges and banks, negative interest rates etc.
- **Control of Elections:** Wrong results (intentional or non-intentional).
- **Control of Jails:** Technology-aided escape attempts and successes, accidental release of inmates, failures in software controlled locks.
- **Law Enforcement:** False arrests and imprisonments.

Software Testing – Introduction

Examples:

- **February 2020: Heathrow disruption**
  - More than 100 flights to and from London's Heathrow airport were disrupted on 16 Feb, 2020, after it was hit by **technical issues affecting departure boards and check-in systems**, leaving passengers with little information about their flights and limiting the use of electronic tickets.
- **August 2019: British Airways**
  - **System failures** caused more than 100 flights to be cancelled and more than 200 others to be delayed. The incident **affected one system for online check-ins** and **another for flight departures**, forcing the airline to revert to manual check-in procedures, leading to long queues at Heathrow, London City airports.
- **Facebook, Instagram and WhatsApp**
  - In the first week of July 2019, users across the globe found themselves **unable to load photos** in the Facebook News Feed, view stories on Instagram, or send messages in WhatsApp.

Software Testing – Introduction

Examples:

- **O_2 - December 2018**
  - More than 30 million O_2 users in the UK **lost access to data services** after a **software issue** left them **unable to use 3G & 4G services.**
- **TSB Bank United Kingdom- April 2018**
  - Millions of TSB **customers were locked out** of their accounts after an **IT upgrade** led to an online banking outage.
- **WannaCry - May 2017 (WannaCrypt0r and Wcry)**
  - Worldwide cyberattack targeted computers running the Microsoft Windows operating system **by encrypting data and demanding ransom payments in the Bitcoin cryptocurrency.**

Software Testing – Introduction

VIT-AP UNIVERSITY

Examples:

- April 2015, Bloomberg terminal in London **crashed due to software glitch** affected more than 300,000 traders on financial markets. It forced the government to postpone a **3billion pound debt sale**.
- Nissan cars **recalled over 1 million cars** from the **market due to software failure in the airbag sensory detectors**. There has been reported two accident due to this software failure.
- **Starbucks** was forced to close about 60 percent of stores in the U.S and Canada due to **software failure in its POS system**. At one point, the store served coffee for free as they were unable to process the transaction.
- Some of Amazon's third-party retailers saw their **product price is reduced to 1paise due to a software glitch**. They were left with **heavy losses**.

Software Testing – Introduction

Examples:

- In 2015 fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.
- In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history
- In May of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.
- China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 innocents live
- In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving 3 people dead and critically injuring 3 others.

Have you heard of any other software bugs?

In the media?

From your personal experience?

Software Testing – Introduction

Most bugs are not because of mistakes in the code …

- Even **experienced programmer make many errors**
  - Avg. 50 bugs per 1000 lines of source code
- **Extensively tested** software contains
  - About 1 bug per 1000 lines of source code
- **Bug origin distribution**
  - Specification (~= 55%)
  - Design (~= 25%)
  - Code (~= 15%)
  - Other (~= 5%)

Relative cost of bugs: "bugs found in later stages - costs more to fix"

Cost to fix a bug increases exponentially (10^x)

i.e., increases tenfold as time increases

E.g.,

a bug found during specification costs $1 to fix.

… if found in design cost is $10

… if found in code cost is $100

… if found in released software cost is $1000

Software Testing – Introduction

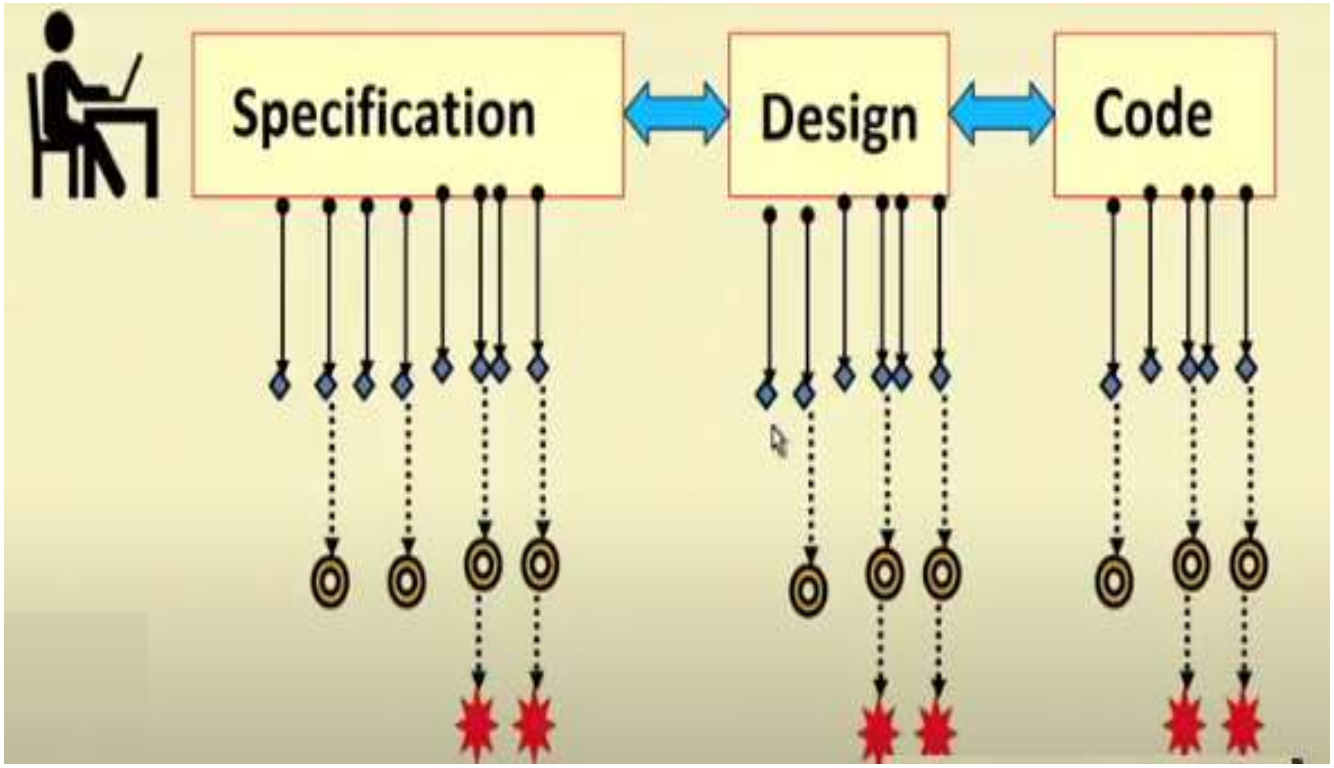
- Software's are **designed** and **developed** by **human beings**
  - **Hence, inherently prone to errors.**
  - **How many? What kind?**
    - Hard to predict the future, however…it is highly likely, that the software to be developed in the future will not be significantly better.
- **If Unchecked**, can lead to a **lot of problems**, including **social implications**.
- **Software Testing** becomes an **essential part** of the software development lifecycle.
- Carrying out the **testing activities for projects** has to be practiced with **proper planning** and must be **implemented correctly.**
- Some of the **questions to be answered when you develop** a software testing strategy.
  - How do you **conduct the tests**? Should you **develop a formal plan** for your tests?
  - Should you **test the entire program as a whole** or **run tests only on a small part** of it?
  - Should you **rerun tests** you've already conducted as you **add new components** to a large system?
  - When should you **involve the customer**?

Software Testing –Introduction

- **Who does Testing?**
  - Project manager, software engineers, and testing specialists.
- **Why is it important?**
  - Accounts for more project effort than any other software engineering action.
- **What are the steps?**
  - Begins “in the small” and progresses “to the large.” As errors are uncovered, they must be diagnosed and corrected using a process called as **debugging**.
- **What is the work product?**
  - Test Specification document.
- **How to ensure that testing done right??**
  - By **reviewing** the Test Specification prior to testing, assess the completeness of test cases and testing tasks.
  - An **effective test plan and procedure** will lead to the orderly construction of the software and the discovery of errors at each stage in the construction process.

Software Testing – Introduction

- The faults or bugs are caused by **mistakes or errors** on the part of the programmer and these in turn may cause failures.



- Let us say the programmer does the specification, design and code.
- Programmer can commit mistakes or errors (can be many during specification, design and code)
- But not all mistakes are fault, defect or bug.
- Example: Programmer written i < 500 instead of i < 50. Its an error.

Some of errors will become fault, Some of faults will become failure

Software Testing – Introduction

VIT-AP UNIVERSITY

- **How to reduce bugs?**
  - Reviews [Specification, Design, code etc.]
    - Effective way
  - **Testing**
    - Widely practiced and acknowledged to be a good technique to reduce bugs.
  - Formal Specification and verification
    - Not used for all the part of code, expensive and difficult to use for larger programs
  - Use of proper development process

VIT-AP UNIVERSITY

How to test?

- Input test data to the program
- Observe the output
  - Check output **matches** the **expected result**
  - If matches, test case passed other wise failed,
  - Under fail case, note down for which input or condition it was failed.
- **Test Report**
- **Based on Test Report, later debug and correct**

Software Testing – Introduction

VIT-AP UNIVERSITY

Testing facts

- Consumes the **largest effort** among all development activities
  - Largest man power among all roles
  - Implies more job opportunities
- Typical estimate is **50% of development effort** spent on testing
  - But **10% of development time**?
  - How?
    - **Parallelism in testing**
    - **Less parallelism** in **specification and design**, because work **is dependent on others**
- Over the years Testing is getting more complex and sophisticated
  - Larger and more complex programs
  - Newer programming paradigms
  - Newer testing techniques
  - Test automation

Software Testing – Introduction

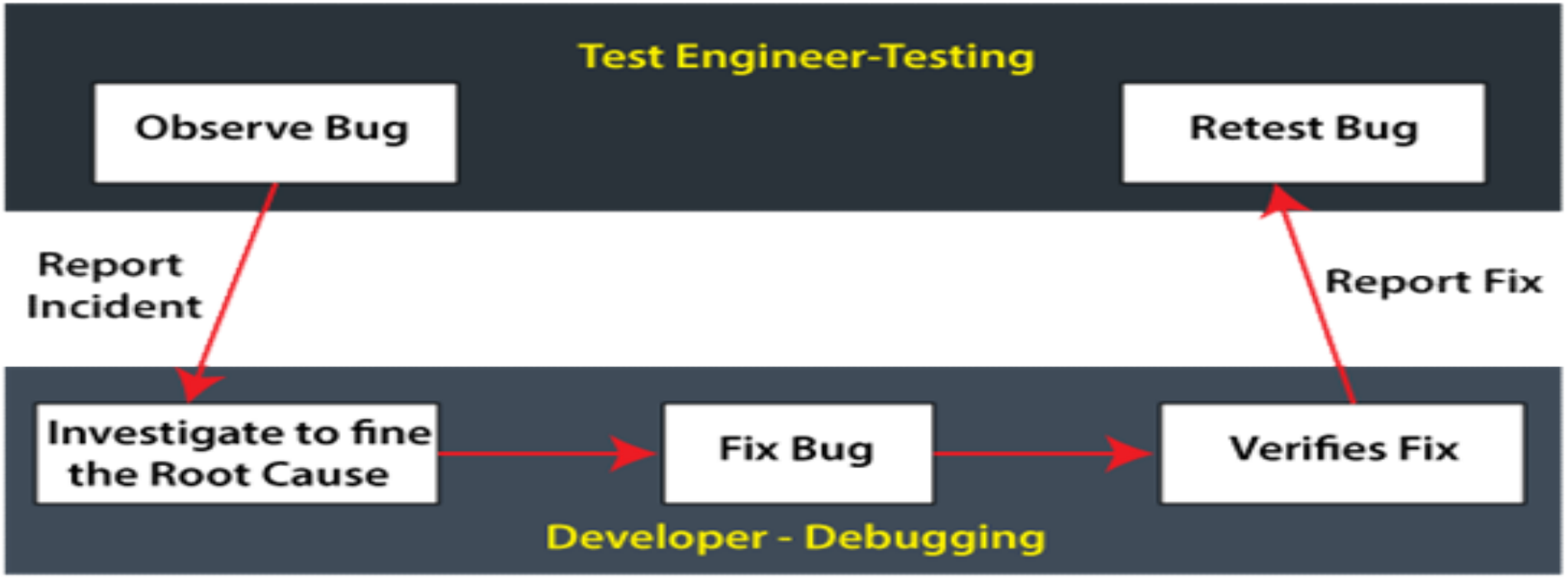
Benefits of Software Testing:

- **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps to save money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product quality:** Essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.

Software Testing – Introduction

Difference Between Testing and Debugging

- Testing and Debugging seems to share a similar meaning but intensively different from one another.



Software Testing – Introduction

Difference Between Testing and Debugging

- **Software testing :**
  - **Process of identifying defects** in the software product.
  - **Performed to validate** the **behavior of the software compared to requirements** before delivering to the clients.
- **Debugging:**
  - **The development team** or a developer **implements an action** after **receiving** the **test report** related to the bugs in the software **from the testing team**.
  - The developer needs to **identify the reason** behind the particular bug or defect, which is **carried out by analyzing the coding rigorously.**
  - Once the debugging is successfully finished, the application is again **sent back** to the test engineers, who remain in the process of testing.

Software Testing – Introduction

Difference Between Testing and Debugging

VIT-AP UNIVERSITY

S.No	Testing	Debugging
1.	It is the implementation of the software with the intent of identifying the defects	The process of fixing and resolving the defects is known as debugging.
2.	Testing can be performed either manually or with the help of some automation tools.	The debugging process cannot be automated.
3.	A group of test engineers executes testing, and sometimes it can be performed by the developers.	Debugging is done by the developer or the programmer.
4.	The test engineers perform manual and automated test cases on the application, and if they detect any bug or error, they can report back to the development team for fixing.	The developers will find, evaluates, and removes the software errors.
5.	Programming knowledge is not required to perform the testing process.	Without having an understanding of the programming language, we cannot proceed with the debugging process.
6.	Once the coding phase is done, we proceed with the testing process.	After the implementation of the test case, we can start the Debugging process.

Software Testing – Introduction

Difference Between Testing and Debugging

S.No	Testing	Debugging
7.	Software Testing includes two or more activities such as validation and verification of the software.	Debugging tries to match indication with cause, hence leading to the error correction.
8.	It is built on different testing levels such as Unit Testing, Integration Testing, System Testing, etc.	It is built on different kinds of bugs because there is no such level of debugging is possible.
9.	Software testing is the presentation of defects.	It is a logical procedure.
10.	Software testing is the vital phase of SDLC (Software Development Life Cycle).	It is not a part of SDLC because it occurs as a subset of testing.
11.	• Software testing contains various type of testing methods, which are as follow: • Black-box testing, White-box testing • Testing types: • Unit testing, Integration Testing • System Testing, Stress Testing • Performance Testing, Compatibility Testing • Beta Testing, Alpha Testing • Smoke Testing, Regression Testing • User Acceptance Testing and so on.	• Debugging involves a various type of approaches, which are as follows: • Induction • Brute Force • Deduction

Software Testing – Verification and Validation (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.**

Verification: **"Are we building the <u>product right</u>?"**
Validation: **"Are we building the <u>right product</u>?"**

When you want to buy a shirt

- Has it got the right number of sleeves,
- is it the right size,
- Is it the right color,
- Does it have all of the buttons?

These are all things that you can verify.

Verification

- Does it fit.
- Is it comfortable to drive in?
- Does the color matches the eyes?
- Can I afford it?
- Is it good quality?
- Will my date like it?

Validation

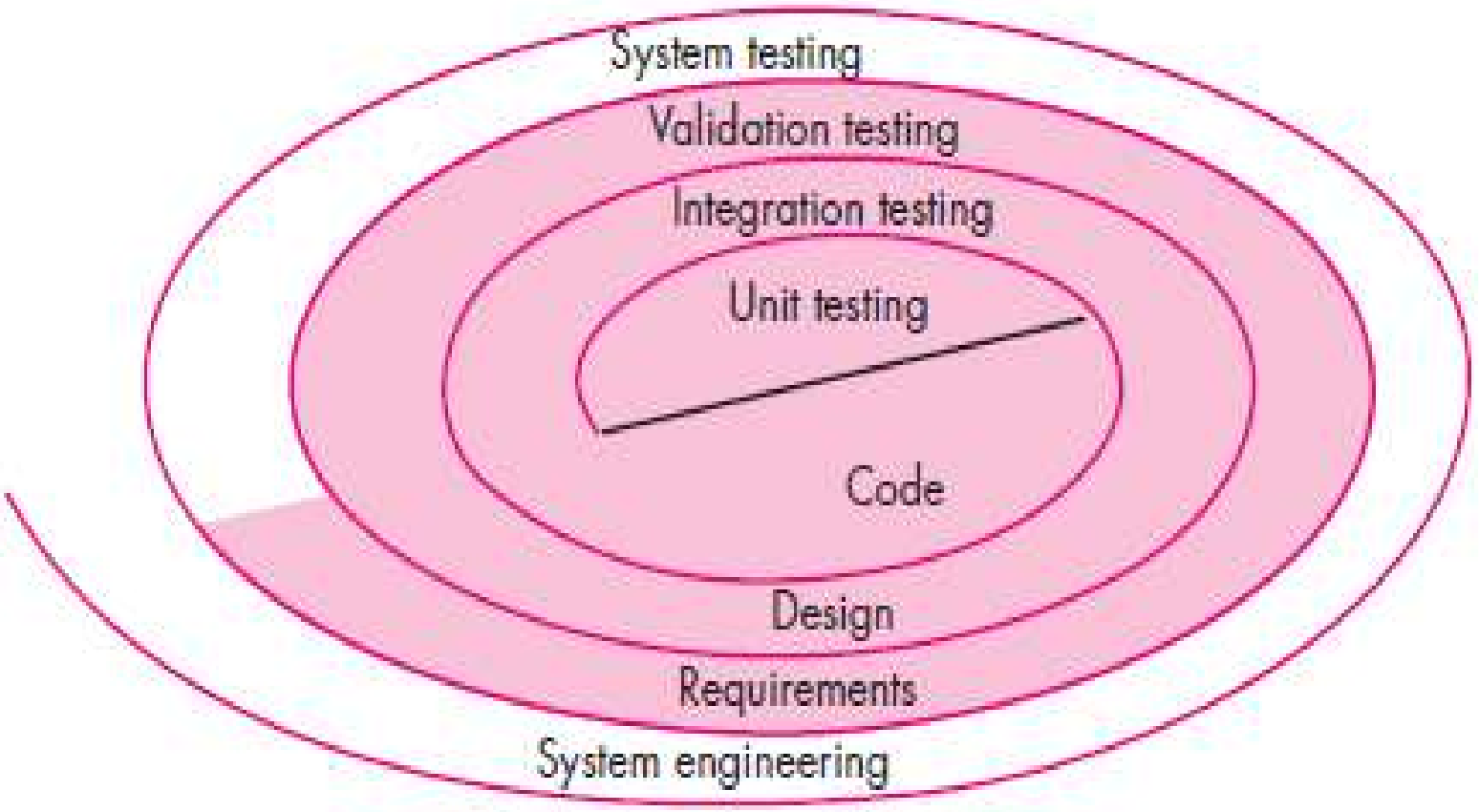
Software Testing – Verification and Validation (V & V)

Difference between Verification and Validation

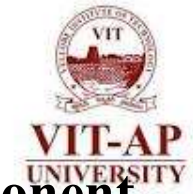
Verification	Validation
Process to find whether the software meets the specified requirements for particular phase.	Process is checked whether the software meets requirements and expectation of the customer.
It estimates an intermediate product.	**It estimates the final product.**
The objective is to check whether software is constructed according to requirement and design specification.	The objective is to check whether the specifications are correct and satisfy the business need.
Describes whether the outputs are as per the inputs or not.	**Explains whether they are accepted by the user or not.**
Done before the validation.	Done after the verification.
Plans, requirement, specification, code are evaluated during the verifications.	Actual product or software is tested under validation.
It manually checks the files and document.	It is a computer software or developed program based checking of files and document.

Software Testing Strategy - The Big Picture

A strategy of software testing is shown in the context of spiral.



Software Testing Strategy - The Big Picture



Unit testing:

- Starts at the **center (vortex)** of the spiral and **concentrates on each unit (e.g., component, class)** of the software as **implemented** in source code

Integration testing:

- Testing progresses by moving outward along the spiral to integration testing
- **Focuses** on the **construction and design** of the software

Validation testing:

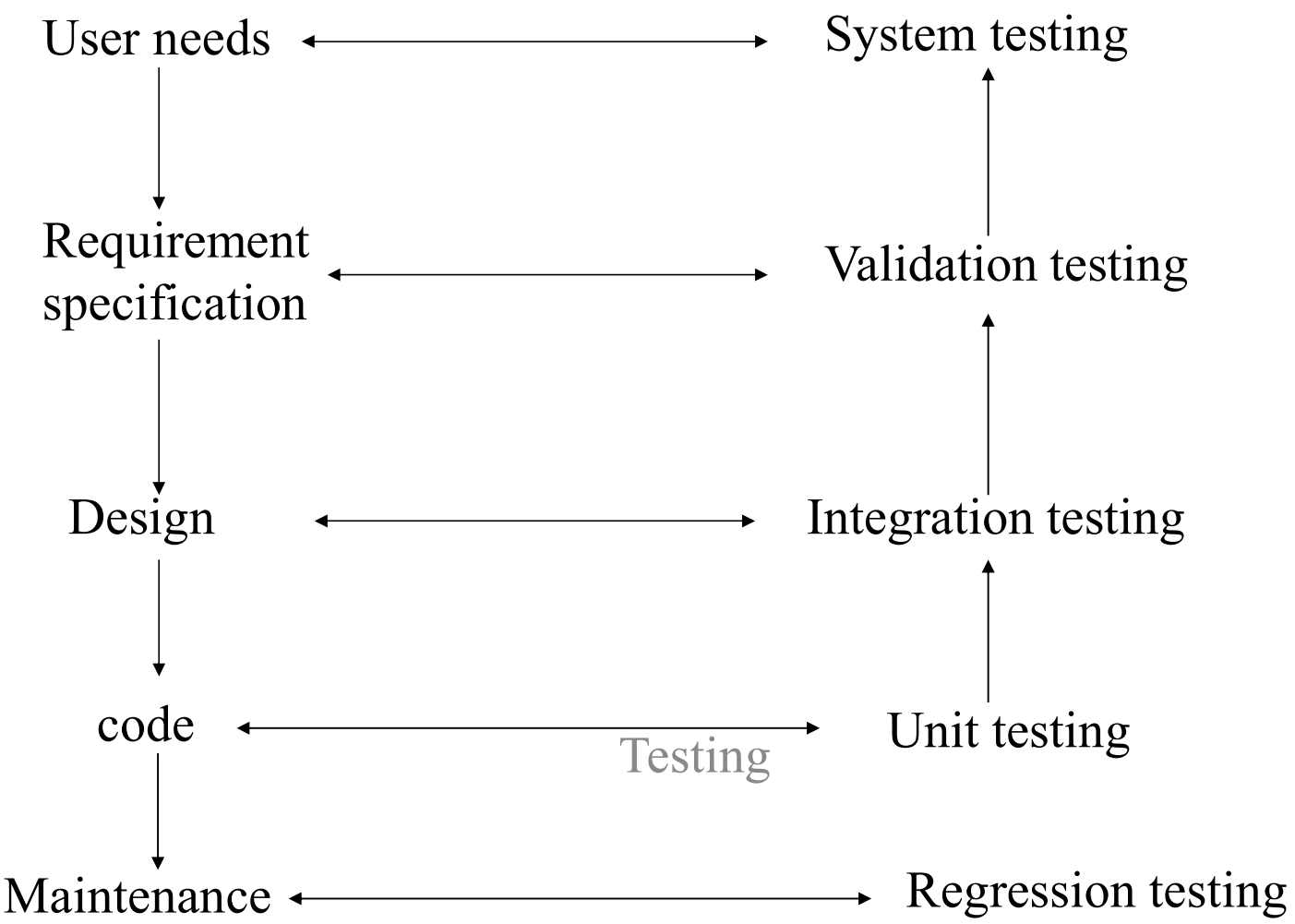
- **Requirements** are **validated** against the software that has been constructed.

System testing:

- Confirms **all system elements** (e.g., hardware, people, databases) and **performance** are tested entirely.

- To test computer software, spiral out in a **clockwise direction** along streamlines that broaden the scope of testing with each turn.

Software Testing Strategy - The Big Picture



Software Testing Strategy – Software Testing Steps

- Testing is a **series of four steps** which are implemented **sequentially**.

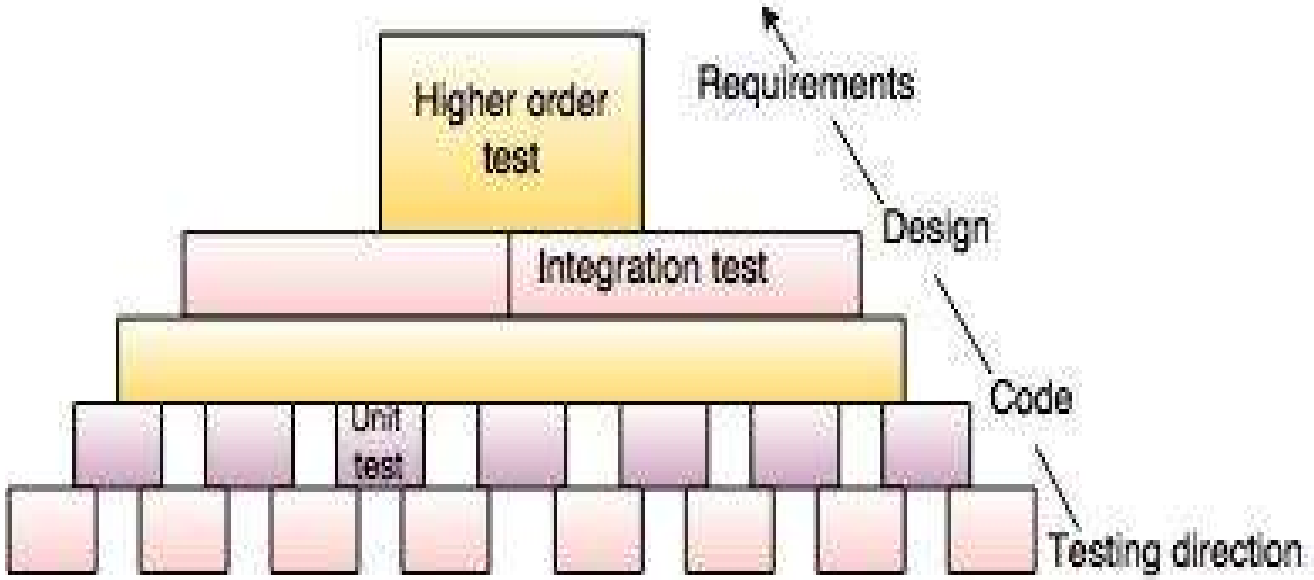


Fig.- Steps of software testing

1. Unit testing
2. Integration testing
3. Validation testing
4. High-order tests

Software Testing Strategy – Guidelines

Software testing strategy will succeed when software testers:

- **Specify product requirements** in a **quantifiable manner** long **before testing commences** i.e., measurable so that testing results are unambiguous
- **State testing objectives explicitly -** test effectiveness, test coverage, meantime-to-failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours should be stated within the test plan.
- **Understand the users of the software** and **develop a profile** for each user category.
- **Develop a testing plan that emphasizes "rapid cycle testing"** - The **feedback** generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

Software Testing Strategy – Guidelines

Software testing strategy will succeed when software testers:

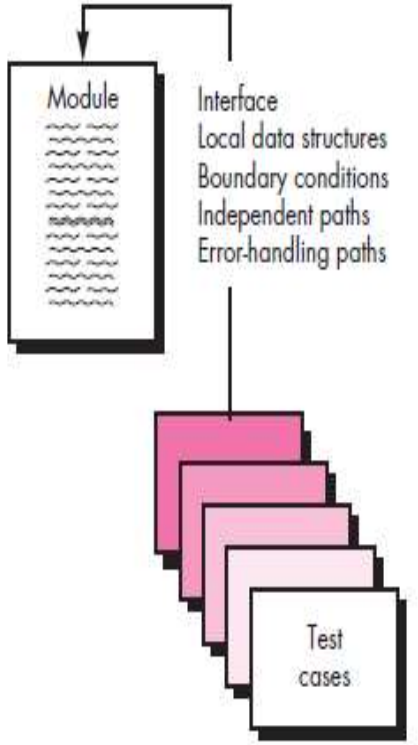
- **Build "robust" software that is designed to test itself** - Software should be designed in a manner that uses antibugging techniques, i.e., software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.
- **Use effective technical reviews as a filter prior to testing -** reviews can reduce the amount of testing effort that is required to produce high quality software.
- **Conduct technical reviews to assess the test strategy and test cases themselves –** to uncover inconsistencies, omissions, and outright errors in the testing approach, which saves time and improves product quality.
- **Develop** a **continuous improvement approach** for the testing process

Test Strategies for Conventional Software - Unit testing

VIT-AP UNIVERSITY

1) Unit testing

- **Focus** on the **smallest unit** of software design, i.e. module or software component.
- The **module interface** is tested to ensure that information properly flows into and out of the program unit under test.
- **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- **All independent paths** through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- **Boundary conditions** are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- Finally, all error-handling paths are tested.



The concept of stubs and drivers:

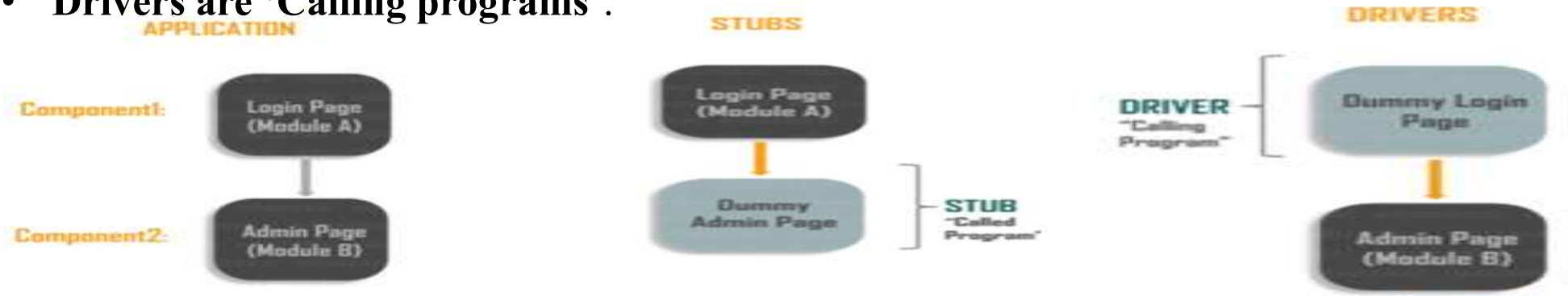
- An application has two modules i.e., ***Login Page (Module A)*** and ***Admin Page (Module B)***.

Case-1:

- To **test the Login Page** which is developed and sent to the testing team.
- Login Page is dependent on Admin Page. But the Admin Page is not ready yet.
- To overcome this situation developers write a dummy program **(Stub)** which acts as an Admin Page. **Stubs are 'Called Programs'.**

Case2:

- To **test Admin Page** but the Login Page is not ready yet.
- To overcome this situation developers write a dummy program (**Driver**) which acts like the Login Page.
- **Drivers are 'Calling programs**'.



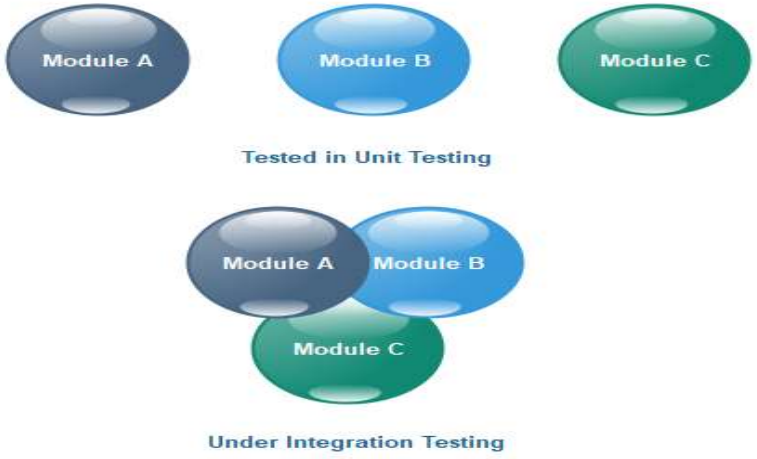
- While testing, situations may arise where some of the modules are still under development. These modules for testing purpose are replaced with some **dummy programs**. These dummy programs are called **stubs and drivers.**

Test Strategies for Conventional Software - Integration testing

VIT-AP UNIVERSITY

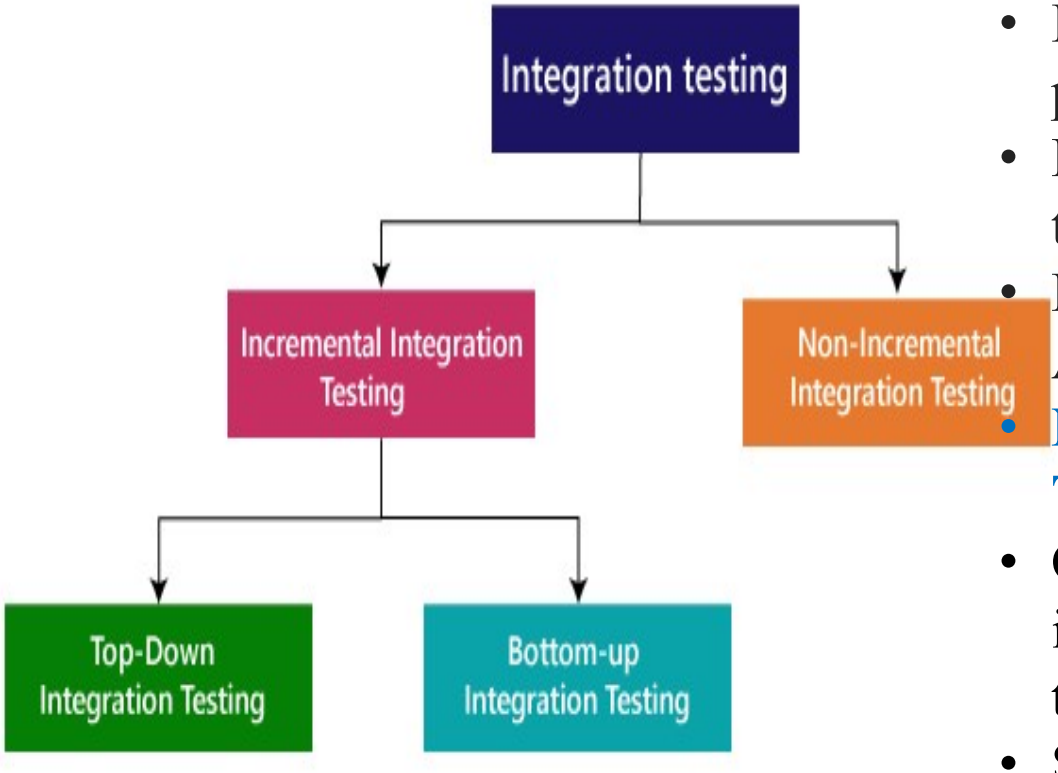
2) Integration testing

- Individual units are combined and tested to verify if they are working as they intend to when integrated.
- The **focus** of the integration testing level is to **expose defects at the time of interaction (interface) between integrated components or units.**



Test Strategies for Conventional Software - Integration testing

Types of Integration Testing

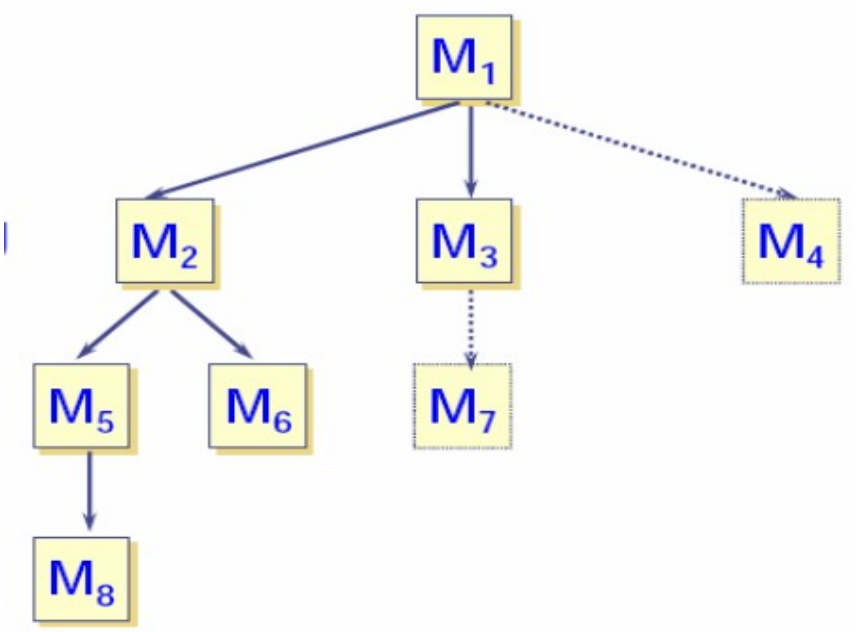


- **Incremental Testing:**
- Performed by connecting two or more modules together that are logically related.
- Later more modules are added and tested for proper functionality.
- Repeated until all the modules are integrated and tested successfully.
- Divided into Top-Down Approach, Bottom-Up Approach.
- **Non Incremental testing (Big Bang Integration Testing)**
- Once all the modules are developed and tested individually, they are integrated once and tested together at once.
- Suitable for smaller systems.

Test Strategies for Conventional Software - Integration testing

Types of Integration Testing – Top down Integration Testing

- Starting with the main control module, the **modules are integrated by moving downward through the control hierarchy**
- Sub-modules to the main control module are incorporated into the structure either in a **depth-first or breadth-first manner**.



Top-Down Testing with Depth-First

- M1, M2, M5, M8
- M6
- M3, M7
- M4

Top-Down Testing with Breath-First

- M1
- M2, M3, M4
- M5, M6, M7
- M8

Test Strategies for Conventional Software - Integration testing

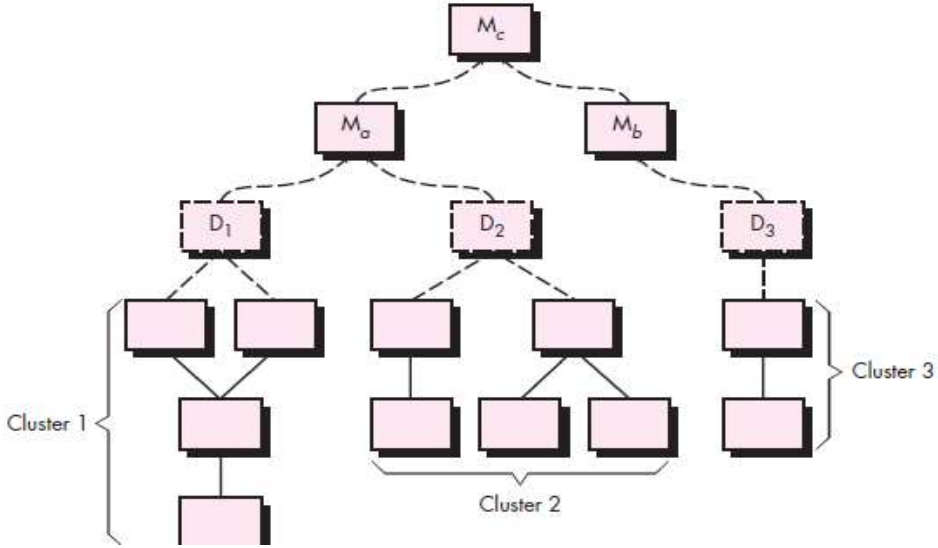
Types of Integration Testing – Top down Integration Testing

- The module integration process:
  - The **main control module** is used as a test driver, and the stubs are substituted for all modules directly subordinate to the main control module.
  - The **subordinate stubs are replaced** one at a time with **actual modules** depending on the approach selected (breadth first or depth first).
  - Tests are executed as each module is integrated.
  - On completion of each set of tests, another **stub is replaced with a real module** on completion of each set of tests
  - To make sure that new errors have not been introduced regression testing may be performed.

Test Strategies for Conventional Software - Integration testing
Types of Integration Testing – Bottom up Integration Testing

VIT-AP

- **Begins** construction and **testing with modules at the lowest level** in the program structure - the modules are integrated from the **bottom to the top.**
- **Steps**
  - **Low-level modules** are combined into **clusters** that perform a specific software sub-function.
  - A driver is written to coordinate test case input and output.
  - The cluster or build is tested.
  - Drivers are removed, and clusters are combined moving upward in the program structure.



- Components are combined to form clusters 1, 2, and 3.
- Each of the clusters is tested using a driver **(a dashed block).**
- Components in clusters 1 and 2 are subordinate to M*a*.
- Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a*.
- Similarly, driver D3 for cluster 3 is removed prior to integration with module M*b*.
- Both M*a* and M*b* will ultimately be integrated with component M*c*, and so forth.

Test Strategies for Conventional Software – Regression Testing

VIT-AP UNIVERSITY

- Each time a new module is added as **part of integration testing**, the **software changes**. New data flow paths are established, new I/O may occur, and new control logic is invoked. These **changes may cause problems with functions that previously worked flawlessly.**
- In the context of an integration test strategy,
  - **Regression Testing** is done to verify that a code change in the software does not impact the existing functionality of the product.
  - It helps to **ensure that changes** (due to testing or for other reasons) **do not introduce unintended behavior or additional errors.**
- The regression test suite (the subset of tests to be executed) contains **three different classes of test cases:**
  - A **representative sample of tests** that will exercise all software functions.
  - **Additional tests** that **focus** on software functions that are **likely to be affected by the change.**
  - **Tests** that **focus** on the software components **that have been changed**.

Test Strategies for Conventional Software – Smoke Testing

- **Software Build :**
  - A simple computer program which consists of only one source code file, just need to compile and link this one file, to produce an executable file.
  - Where as a typical Software Project consists of **hundreds or even thousands of source code files.**
  - Creating an **executable program** from these source files is a complicated and time-consuming task.
  - In such case use **"build"** software to create an executable program and the process is called ***"Software 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 **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.

Test Strategies for Conventional Software – Smoke Testing

Benefits of Smoke Testing when it is applied on complex, time-critical software projects:

- **Integration risk is minimized -** As smoke tests are conducted daily, incompatibilities and other show-stopper errors (causes an implementation to stop and become essentially useless) are uncovered early, thereby reducing the serious schedule impact when errors are uncovered.
- **The quality of the end product is improved -** Because the approach is construction (integration) oriented, focus is to uncover functional errors, architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- **Error diagnosis and correction are simplified -** Errors uncovered during smoke testing are likely to be associated with "new software increments"- i.e., the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- **Progress is easier to assess -** With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

Test Strategies for Conventional Software – System Testing

- **Performed** on a **complete integrated system** to evaluate the compliance of the system with the **corresponding requirements**.
- The **result** of system testing is the **observed behavior of a component** or a **system** when it is tested.
- **Tests the design and behavior of the system** and also the **expectations of the customer**.
- It is performed to test the system beyond the bounds mentioned in the software requirements specification (SRS).
- **Types of System Testing**
  - Recovery Testing
  - Security Testing
  - Stress Testing
  - Performance Testing
  - Deployment Testing
  - Scalability Testing

Test Strategies for Conventional Software – System Testing

VIT-AP UNIVERSITY

Types of system tests:

Recovery testing:

- *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), reinitialization, 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 protection mechanisms** built into a system and **protect** it from improper penetration.

Test Strategies for Conventional Software – System Testing

VIT-AP UNIVERSITY

Stress Testing:

- Performed to **check the robustness of the system under the varying loads.** (Essentially, the tester attempts to break the program)
- Tests may be **designed that generate**
  - **Ten interrupts per second**, when one or two is the average rate
  - **Input data rates may be increased** by an order of magnitude to determine how input functions will respond,
  - Test cases that require **maximum memory or other resources are executed**,
  - Test cases that **may cause thrashing in a virtual operating system** are designed,
  - Test cases that **may cause excessive hunting for disk-resident data** are created.

Test Strategies for Conventional Software – System Testing

Performance Testing

- Carried out to **test the speed, stability and reliability of the software product or application.**
- Performance testing **occurs** throughout **all steps in the testing process**. At the unit level, the performance of an individual module may be assessed as tests are conducted.
- Performance tests are often **coupled with stress testing** and usually require both hardware and software instrumentation.
- To **measure resource utilization (e.g., processor cycles)** in an exacting fashion.

Test Strategies for Conventional Software – System Testing

Deployment testing (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 (e.g., "installers")** that will be used by customers, and all documentation that will be used to introduce the software to end users.

Scalability Testing:

- Carried out to check the performance of a software application or system in terms of its capability to **scale up or scale down** the **number of user request load.**

Software Testing Fundamentals

- The **goal of testing** is to find errors, and a **good test** is one that has a **high probability of finding an error.**
- **Software testability** is **how easy system or program or product can be tested.**

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

Software Testing Fundamentals

VIT-AP UNIVERSITY

Attributes of 'good' test

- **High probability of finding an error**.
  - A Tester must understand the software and how it might fail.
- **Not redundant**
  - Testing time is limited, one test should not server the same purpose as another test
- **Should be "best of breed"**
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used.
- **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

Software Testing Fundamentals

Testing does not start at the end of software process.
It is performed throughout the software process.
Test case design and test planning can begin as soon as requirement analysis is completed.

VIT-AP UNIVERSITY

Internal and External Views of a Testing:

1. Internal program logic is exercised using **white box testing**.
2. Software requirements are exercised using **black box testing**.

White Box Testing

- Also known as **structural testing, clear box testing, glass box testing, open box testing and transparent box testing**.
- A test-case design philosophy that **uses the control structure described as part of component-level design** to derive test cases.
- Methods of White Box Testing
  1. **Guarantee that all independent paths** within a module have been exercised at least once,
  2. **Exercises all logical decisions** on their true and false sides
  3. **Execute all loops at their boundaries** and **within their operational bounds**, and
  4. **Exercise internal data structures** to ensure their validity.

White Box Testing

- **Programming skills** are required to design test cases as every line of the code of the program is tested
- The primary goal is to **focus on the flow of inputs and outputs** through the software and **strengthening the security** of the software.
- The **developers** perform the **white-box testing** and then send the software to the testing team, where **testing team** will **perform the black box testing** and verify the software along with the requirements and identify the bugs and sends it to the developer.

  The White Box Testing performs following tests:
  - Path testing
  - Loop testing
  - Condition testing
  - Testing based on the memory perspective
  - Test performance of the program

White Box Testing – Basis Path Testing

- Proposed by **Tom McCabe**
- The basis path method 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.**

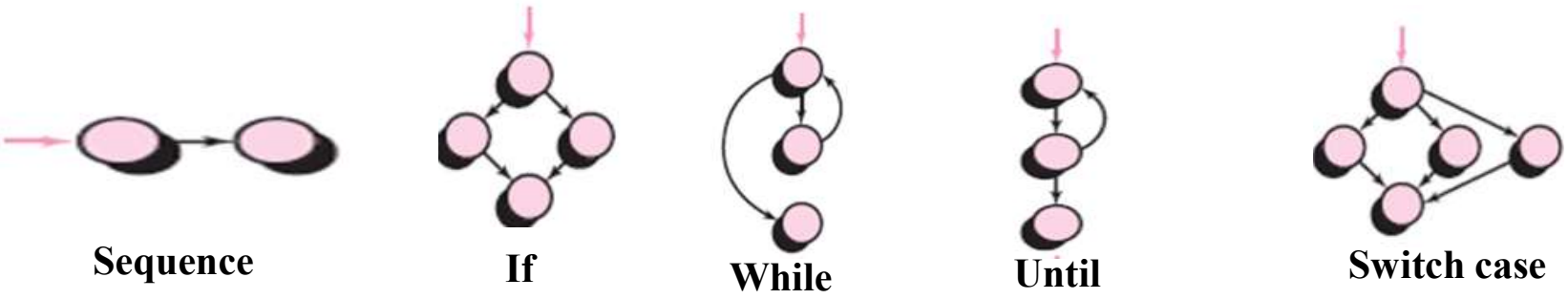
Steps:

- Construct the control flow graph from the program
- Determine the Cyclomatic complexity
- Determine a basis set of linearly independent paths
- Design test cases corresponding to each independent path

White Box Testing – Basis Path Testing

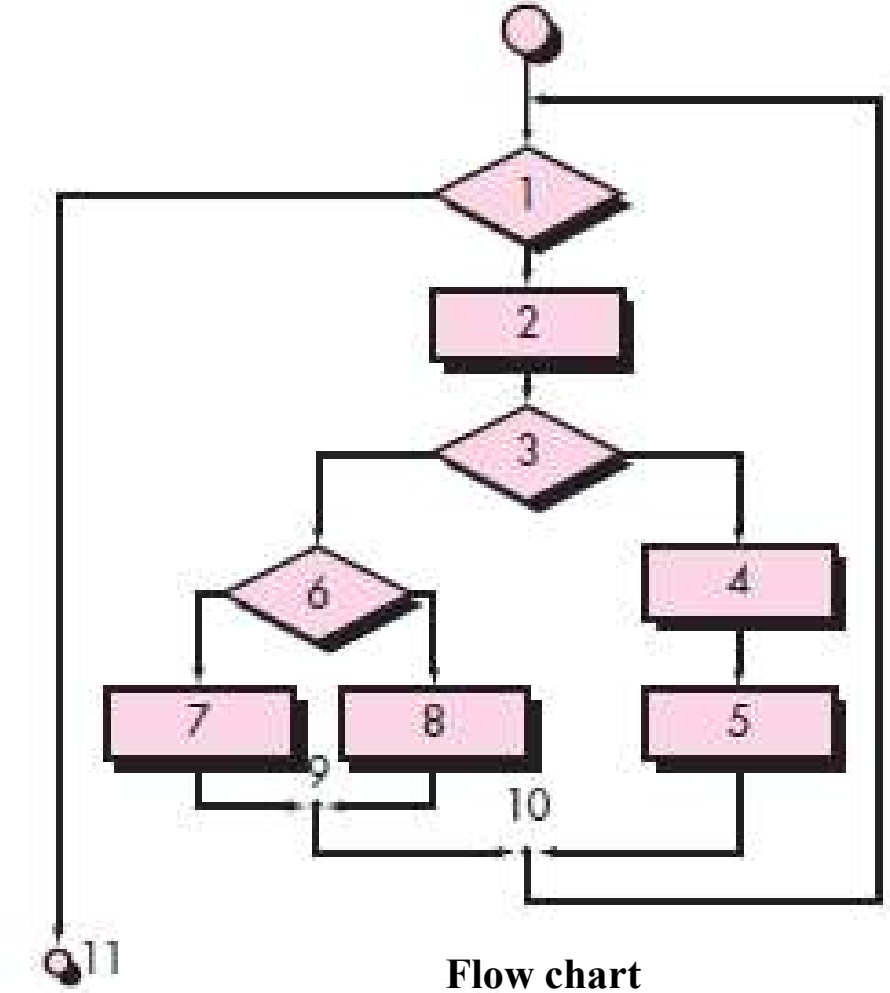
Flow Graph Notation

- The representation of control flow, called a **flow graph (or program graph)**
- The flow graph depicts **logical control flow** using the following notations

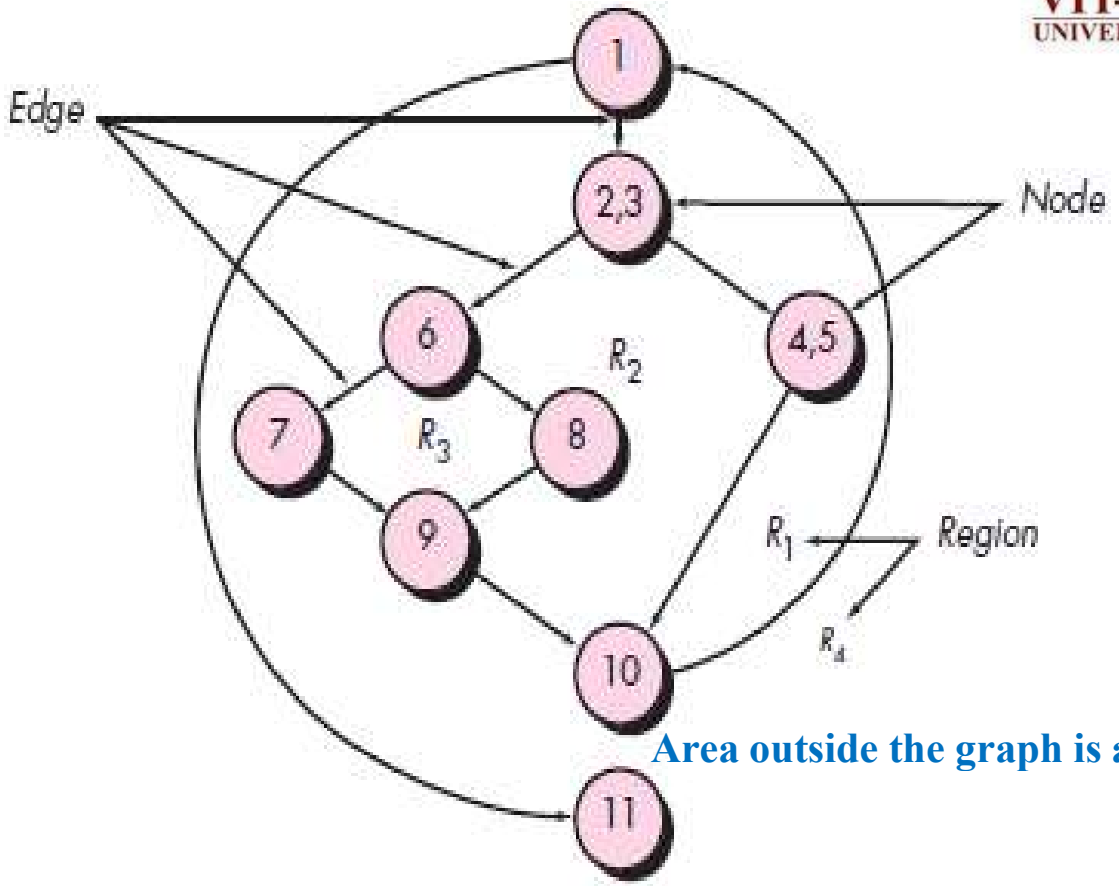


- Each **circle**, called a ***flow graph node***, represents **one or more procedural statements.**
  - ***Predicate node:*** A node that contains a conditional expression and has **two edges leading** out from it.
- The **arrows** on the flow graph, called ***edges*** **or** ***links***, represent **flow of control**
  - An edge must **terminate at a node** and **does not intersect or cross over another edge**
- **Areas bounded by edges and nodes** are called ***regions***.
  - When counting regions, **include** the **area outside the graph as a region.**

White Box Testing – Basis Path Testing



Flow chart



Flow Graph

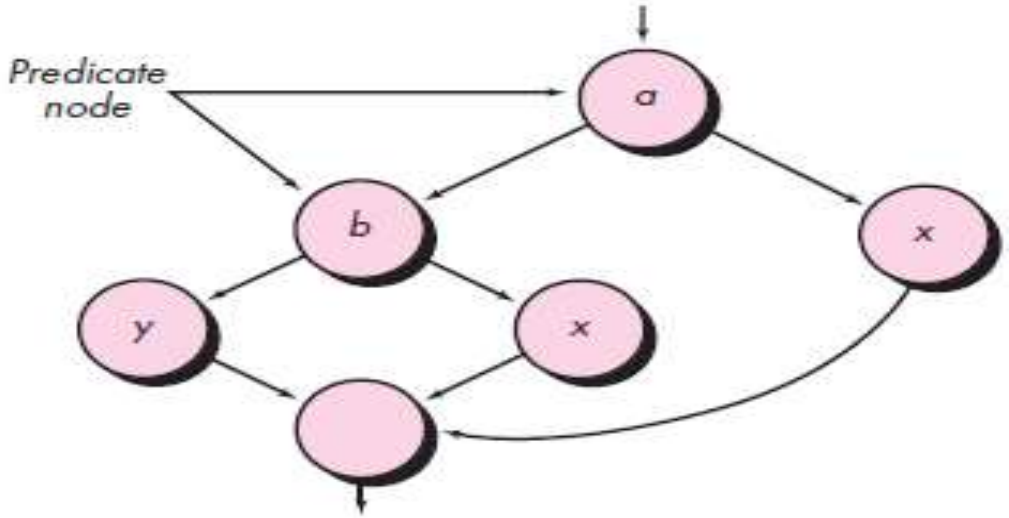
Area outside the graph is a region

- A **sequence of process boxes and a decision diamond can map into** a single node **(2,3).**

White Box Testing – Basis Path Testing

Compound logic

If a OR b then
    Procedure X
Else
    Procedure Y

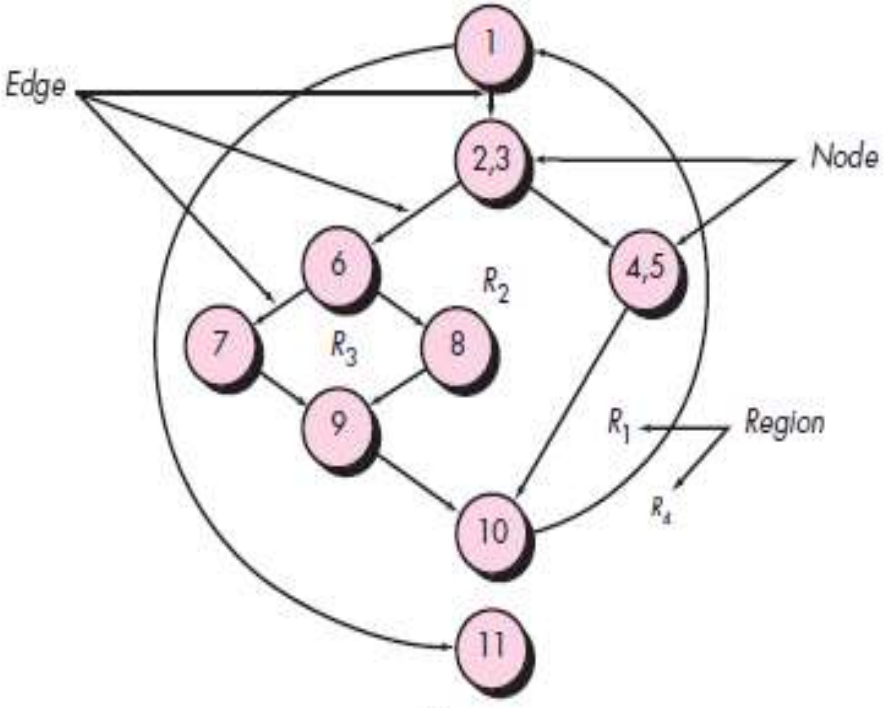


- A **compound condition** occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.
- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated.

White Box Testing – Basis Path Testing

Independent Program Paths

- An ***independent path*** is any path through the program that introduces **at least one new set of processing statements or a new condition**.
- In a flow graph, an independent path must move along **at least one edge that has not been traversed before the path is defined**.



Path 1: 1-11
Path 2: 1-2-3-4-5-10-1-11
Path 3: 1-2-3-6-8-9-10-1-11
Path 4: 1-2-3-6-7-9-10-1-11

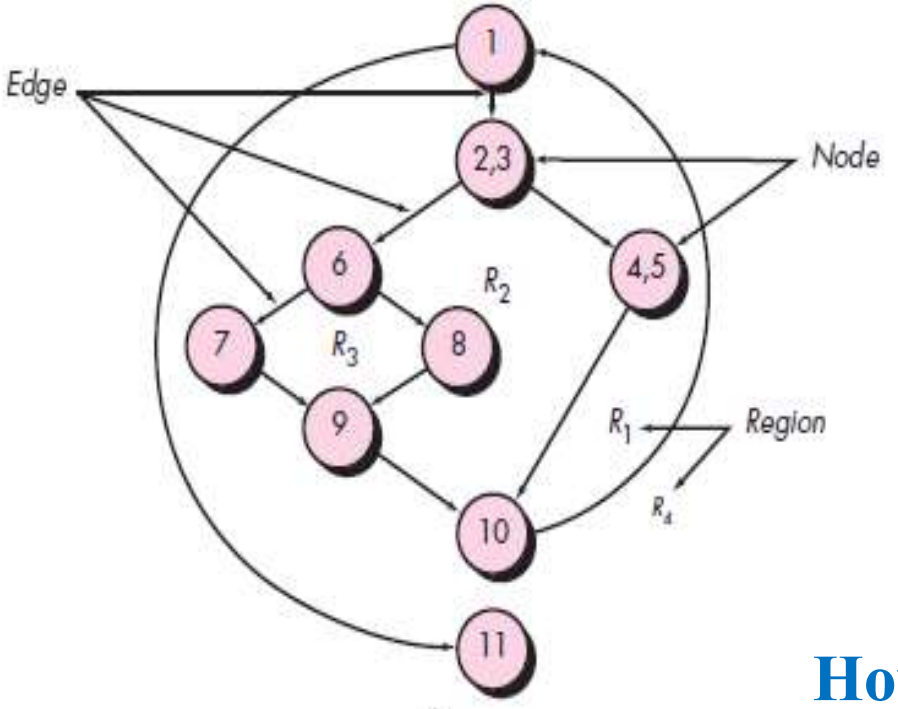
Note that each new path introduces a new edge

Can 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 be considered as an independent path?

No - A combination of already specified paths and does not traverse any new edges.

White Box Testing – Basis Path Testing

Independent Program Paths



Path 1: 1-11
Path 2: 1-2-3-4-5-10-1-11
Path 3: 1-2-3-6-8-9-10-1-11
Path 4: 1-2-3-6-7-9-10-1-11

- Paths 1 through 4 constitute a **basis set** for the flow graph.
- Every **statement** in the program will have been **guaranteed to be executed at least one time** and **every condition** will have been **executed on its true and false sides.**
- It should be noted that the basis set is **not unique**, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for?

Ans: Computation of Cyclomatic Complexity

White Box Testing – Basis Path Testing

Cyclomatic complexity is computed in one of three ways:

Based on edges and nodes:

V(G) = e - n + 2

Where,

e is number of edges

n is number of nodes

Based on Regions :

V(G) = Number of regions in the graph

All the three formulae, the cyclomatic complexity obtained remains same.

Based on Decision Nodes:

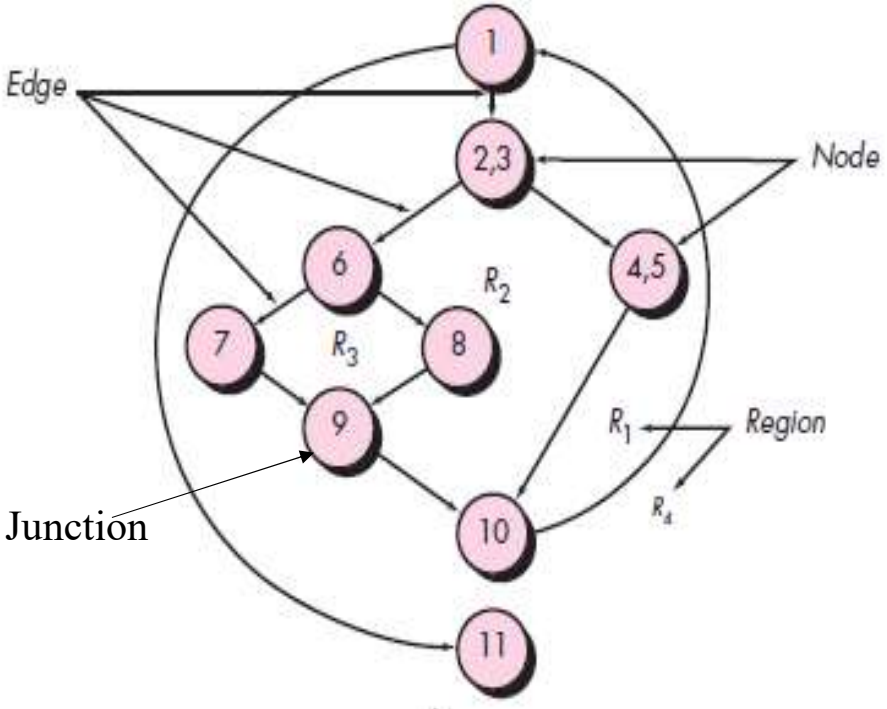
V(G) = P + 1

Where,

P is number of predicate nodes

White Box Testing – Basis Path Testing

Independent Program Paths



Cyclomatic complexity:

1. V(G) = 11 edges - 9 nodes + 2 = 4
2. V(G) = 3 predicate nodes + 1 = 4
3. V(G) = 4 Four regions.

- **Cyclomatic complexity** is a **software metric** that **provides a quantitative measure of the logical complexity of a program**
- Cyclomatic complexity **provides** the **upper bound** on the number of test cases that will be required to guarantee that every statement in the program has been executed at least one time.
- The basis path testing method can be **applied to a Procedural design or to source code**

White Box Testing – Basis Path Testing

- The basis path method 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.**

Steps:

- Using the design or code as a foundation, draw a corresponding flow graph.
- Determine the cyclomatic complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.

White Box Testing – Basis Path Testing

Example - 1: Prime number

```
int main(){
    int n, index;
    cout << "Enter a number: " << endl;
    cin >> n;
    index = 2;
    while (index <= n - 1) {
        if (n % index == 0) {
            cout << "It is not a prime number" << endl;
            break;
        }
        index++;
    }
    if (index == n)
        cout << "It is a prime number" << endl;
} // end main
```

Step-1: Draw a corresponding flow graph:

- Start numbering the statements **after declaration of the variables (if no variables have been initialized in that statement)**.
- However**, if a variable has been initialized and declared in the same line, then numbering should start from that line itself.**

White Box Testing – Basis Path Testing

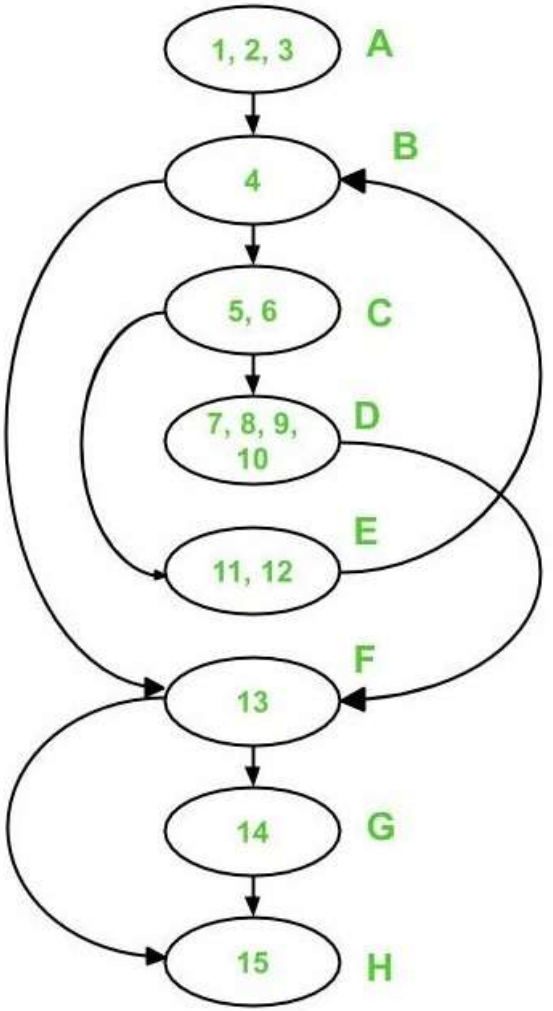
```
int main()
{
   int n, index;
1   cout << "Enter a number: "
2   cin> n;
3    index = 2;
4    while (index <= n - 1)
5    {
6       if (n % index == 0)
7       {
8          cout << "It is not a prime number" << endl;
9          break;
10      }
11      index++;
12   }
13   if (index == n)
14      cout << "It is a prime number" << endl;
15 } // end main
```

Step-1: Draw a corresponding flow graph:

- Put the **sequential statements** into **one single node**.
- Statements 1, 2 and 3 are all sequential statements and hence should be combined into a single node.
- And for other statements, follow the notations as per flow diagram
- Note –
  - **Use alphabetical numbering on nodes for simplicity**.

White Box Testing – Basis Path Testing

```
int main()
{
   int n, index;
1   cout << "Enter a number: "
2   cin> n;
3    index = 2;
4    while (index <= n - 1)
5    {
6       if (n % index == 0)
7       {
8          cout << "It is not a prime number" << endl;
9          break;
10       }
11       index++;
12   }
13   if (index == n)
14       cout << "It is a prime number" << endl;
15 } // end main
```



White Box Testing – Basis Path Testing

Step – 2: Calculate the Cyclomatic complexity:

Method-1: V(G) = e - n + 2

e = 10, n = 8

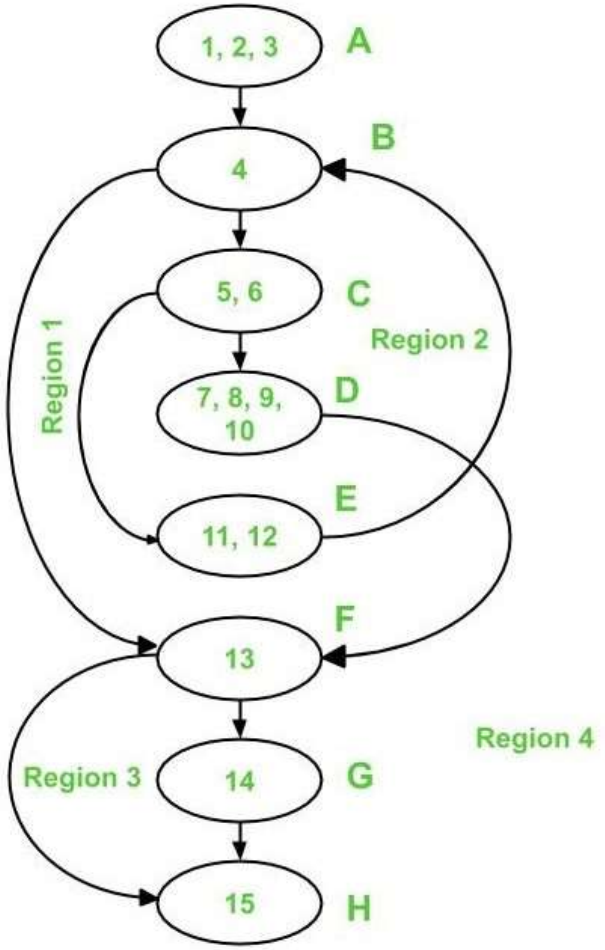
= 10 - 8 + 2 = 4

Method-2: **V(G) = P + 1 = 3 + 1 = 4**

where, P = 3 (Node B, C and F)

Method-3: V(G) = Number of Regions

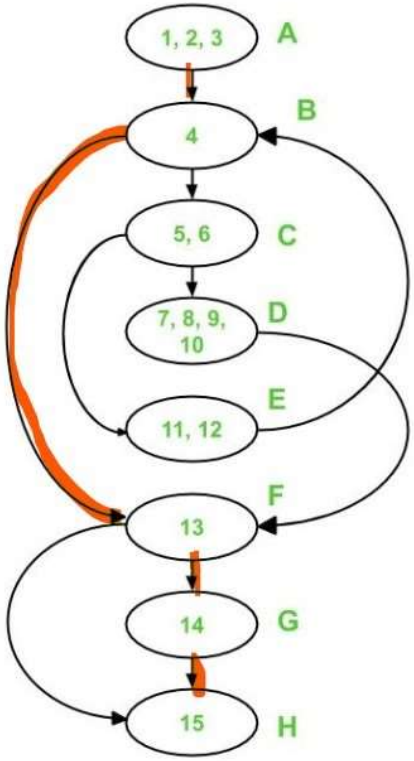
4 regions: R1, R2, R3 and R4



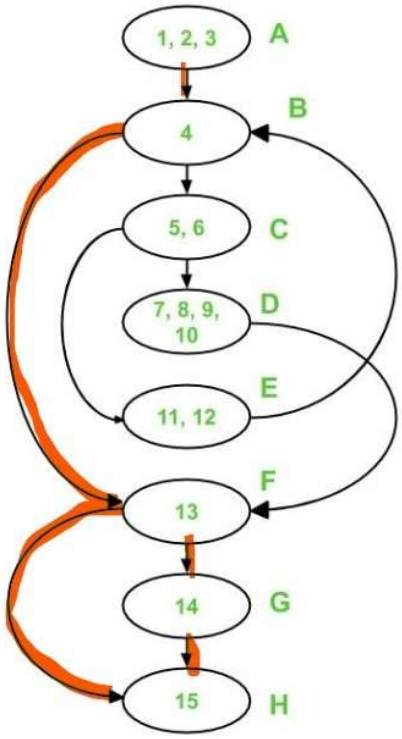
White Box Testing – Basis Path Testing

Step -3 : Determine a basis set of linearly independent paths

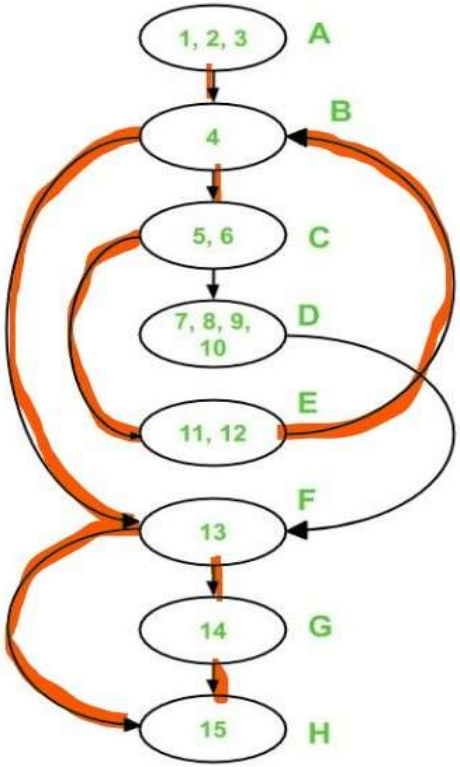
Path 1 : A - B - F - G - H



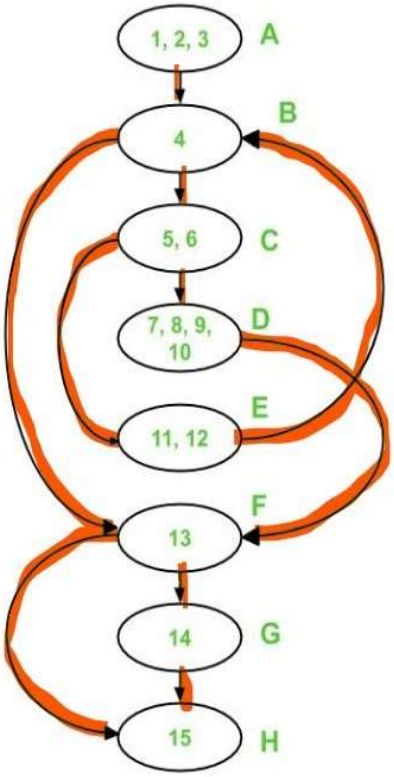
Path 2 : A - B - F - H



Path 3 : A - B - C - E - B - F - G - H



Path 4 : A - B - C - D - F - H



Now only 2 edges are left uncovered i.e. edge C-D and edge D-F.
Hence, Path 4 must include these two edges.

White Box Testing – Basis Path Testing

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

- Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
- Each test case is executed and compared to expected results.
- Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

Test case ID	Input Number	Output	Independent Path covered
1	1	No output	A-B-F-H
2	2	It is a prime number	A-B-F-G-H
3	3	It is a prime number	A-B-C-E-B-F-G-H
4	4	It is not a prime number	A-B-C-D-F-H

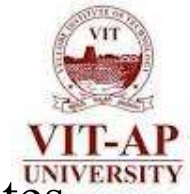
White Box Testing – Basis Path Testing

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

- Statement Testing
  - 100%statement / node coverage
- Branch Testing
  - 100% branch/ link coverage
- Path Testing
    - 100%path coverage

Statement Testing < Branch Testing < Path Testing

White Box Testing – Basis Path Testing

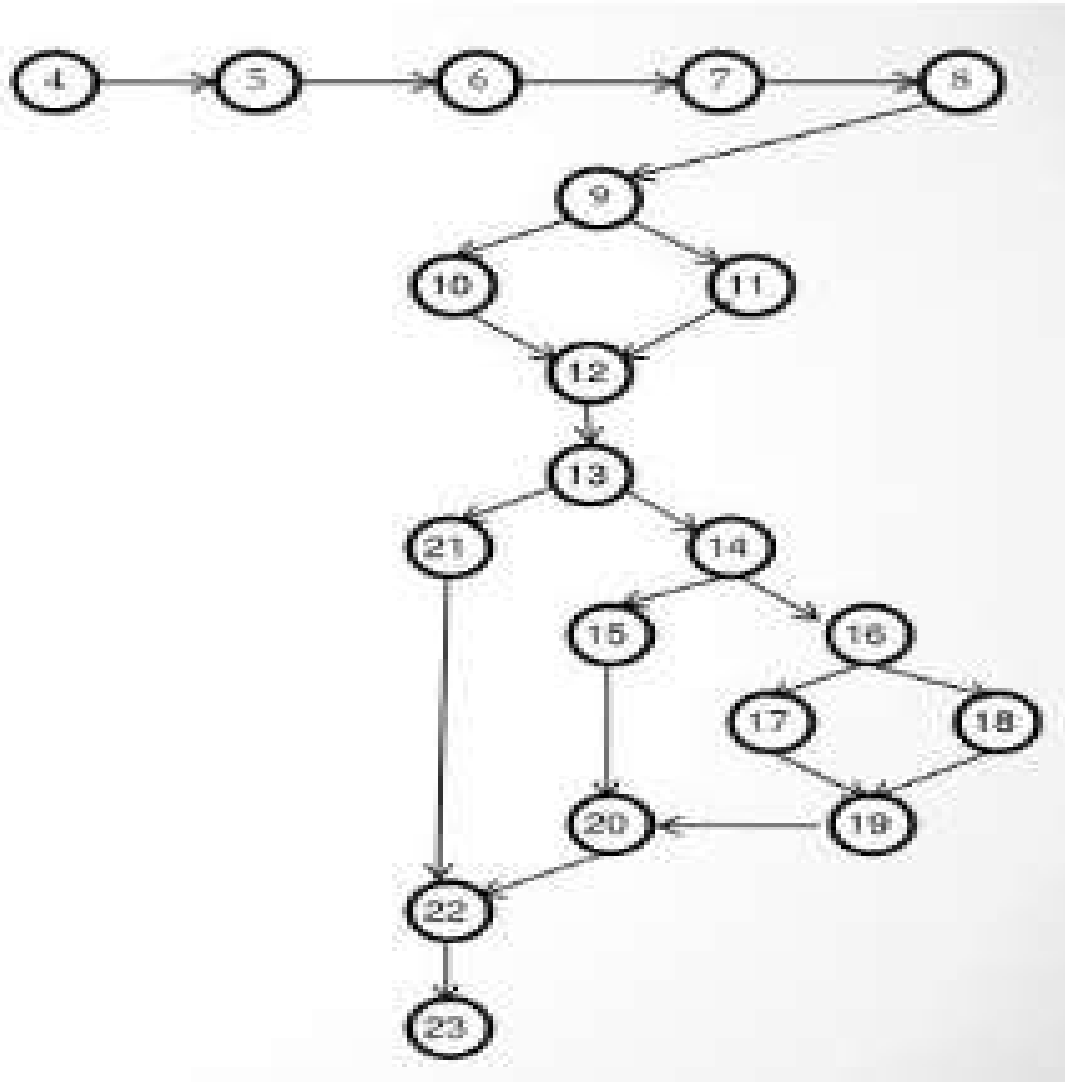


Example- 2 : A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.
Develop a set of test cases that you feel will adequately test this program.

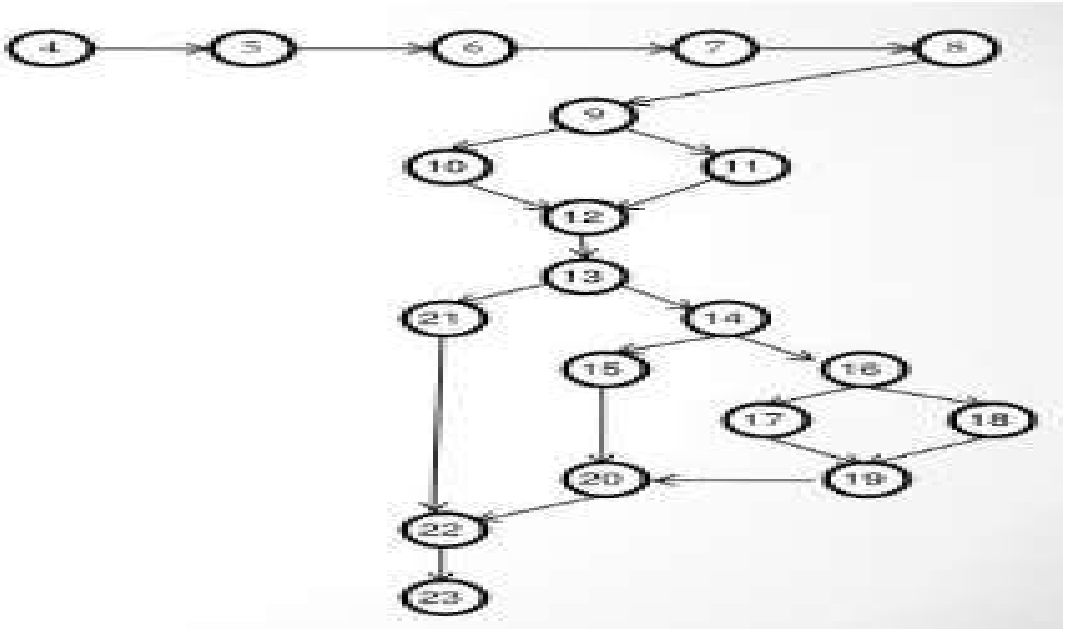
```
1. Program Triangle
2. int a, b, c
3. bool isTraingle
4. write "enter a, b, and c integers"
5. read a,b,c
6. Write("side 1 is", a)
7. Write("side 2 is", b)
8. Write("side 3 is", c)
9. If (a < b + c) AND (b < a+c) AND (c < b + a)
10. then IsTriangle = True
11. else IsTriangle = False
12. endif
13. If IsTriangle then
14.         If (a == b) AND (b == c) then
15.                 Write("Equilateral")
16.         else if (a != b) AND (a !=b) AND (b !=c) then
17.                 Write("Scalene")
18.            else Write("Isosceles")
19.           endif
20.         endif
21.  else Write("not a triangle")
22. endif
23. end Triangle
```

White Box Testing – Basis Path Testing

```
1. Program Triangle
2. int a, b, c
3. bool isTraingle
4. write "enter a, b, and c integers"
5. read a,b,c
6. Write("side 1 is", a)
7. Write("side 2 is", b)
8. Write("side 3 is", c)
9. If (a < b + c) AND (b < a+c) AND (c < b + a)
10. then IsTriangle = True
11. else IsTriangle = False
12. Endif
13. If IsTriangle then
14.         If (a == b) AND (b == c) then
15.                 Write("Equilateral")
16.         else if (a != b) AND (a !=b) AND (b !=c) the
17.                 Write("Scalene")
18.           else Write("Isosceles")
19.          endif
20.         endif
21.  else Write("not a triangle")
22. endif
23. end Triangle
```



White Box Testing – Basis Path Testing



Paths

- 4-5-6-7-8-9-10-12-13-21-22-23
- 4-5-6-7-8-9-11-12-13-14-15-20-22-23
- 4-5-6-7-8-9-11-12-13-14-16-17-19-20-22-23
- 4-5-6-7-8-9-11-12-13-14-16-18-19-20-22-23

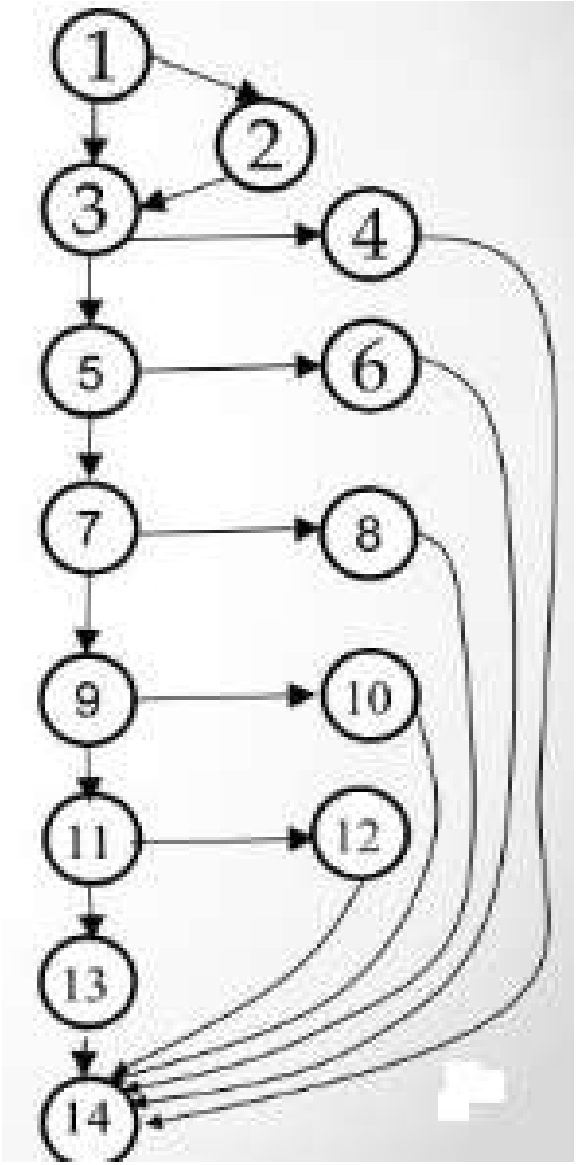
Path	Decision				Test case			Expected Results
	9	13	14	16	a	b	c	
1	T	F			100	100	200	Not a Triangle
2	F	T	T		100	100	100	Equilateral
3	F	T	F	T	100	50	60	Scalene
4	F	T	T	F	100	100	50	Isosceles

White Box Testing – Basis Path Testing

Example-3:

```
public double calculate(int amount)     {
1. double rushCharge = 0;
   if (nextday.equals("yes") ){
2. rushCharge = 14.50; }
3  double tax = amount * .0725;
3  if (amount >= 1000) {
4.  shipcharge = amount * .06 + rushCharge; }
5. else if (amount >= 200) {
6.  shipcharge = amount * .08 + rushCharge; }
7. else if (amount >= 100) {
8. shipcharge = 13.25 + rushCharge; }
9. else if (amount >= 50) {
10. shipcharge = 9.95 + rushCharge; }
11. else if (amount >= 25) {
12. shipcharge = 7.25 + rushCharge; }
else {
13. shipcharge = 5.25 + rushCharge; }
14. total = amount + tax + shipcharge;
14. return total;
} //end calculate
```

Step 1: Draw the flow graph.



VIT-AP UNIVERSITY

White Box Testing – Basis Path Testing

Step 2: Determine the cyclomatic complexity of the flow graph.

V(G) = E - N + 2 → 19 - 14 + 2 → 7

Step 3: Determine the basis set of independent paths.

Path 1: 1 - 2 - 3 - 5 - 7 - 9 - 11 - 13 - 14

Path 2: 1 - 3 - 4 - 14

Path 3: 1 - 3 - 5 - 6 - 14

Path 4: 1 - 3 - 5 - 7 - 8 - 14

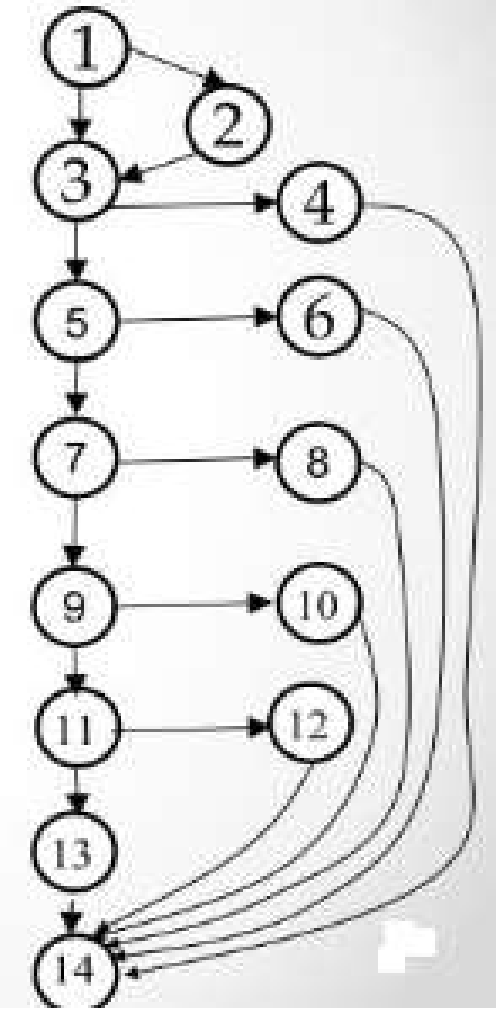
Path 5: 1 - 3 - 5 - 7 - 9 - 10 - 14

Path 6: 1 - 3 - 5 - 7 - 9 - 11 - 12 - 14

Path 7: 1 - 3 - 5 - 7 - 9 - 11 - 13 – 14

Step 4: Prepare test cases that force execution of each path in the basis set.

Path	Nextday	Amount	Expected Results
P1	yes	10	30.48
P2	No	1500	1713.25
P3	No	300	345.75
P4	No	150	174.125
P5	No	75	90.3875
P6	No	30	39.425
P7	No	10	15.975



White Box Testing – Basis Path Testing

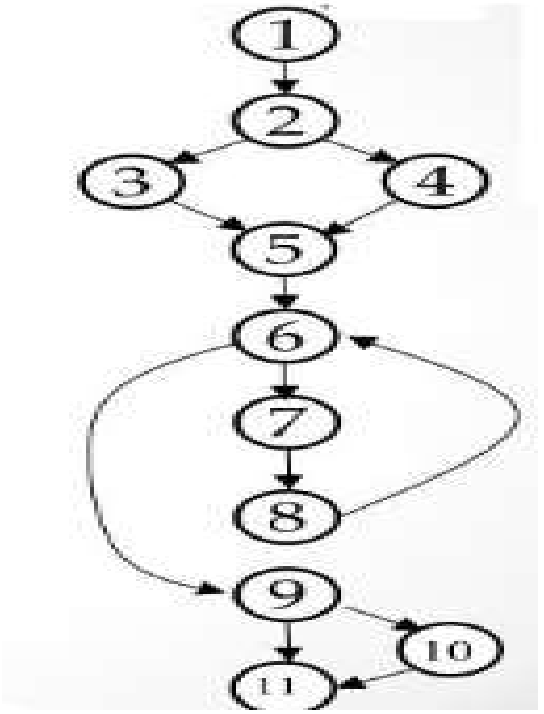
Example - 4:

```
double sample(int x, int y){
        if(y < 0)
                pow = -y;
        else
                pow = y;
        z = 1.0;
        while(pow!=0) {
                z *=x;
                pow = pow – 1;}
        if(y < 0)
                z = 1.0 / z
        return z;} // end sample
```

Example - 4:

```
1  double sample(int x, int y){
2          if(y < 0)
3                  pow = -y;
           else
4                  pow = y;
5          z = 1.0;
6          while(pow!=0) {
7                  z *=x;
8                  pow = pow – 1;}
9          if(y < 0)
10                 z = 1.0 / z
11         return z; } // end sample
```

Step-1: Draw a corresponding flow graph:

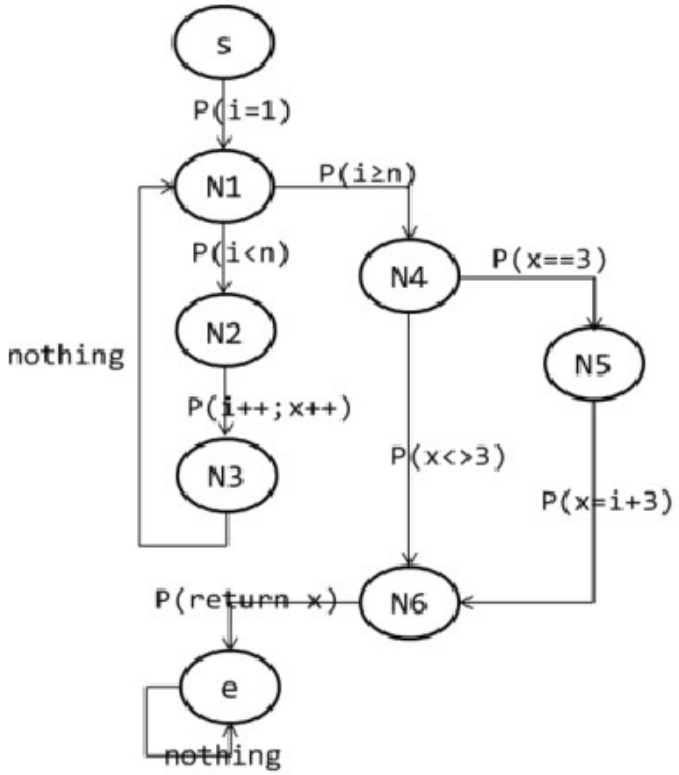


White Box Testing – Basis Path Testing

Example - 5:

```
int sample(int x, int n){
        int i =1;
        while(i<n) {
                i++;
                x++;
        }
        if(x==3)
                x = i + 3;
        return x;
} // end sample
```

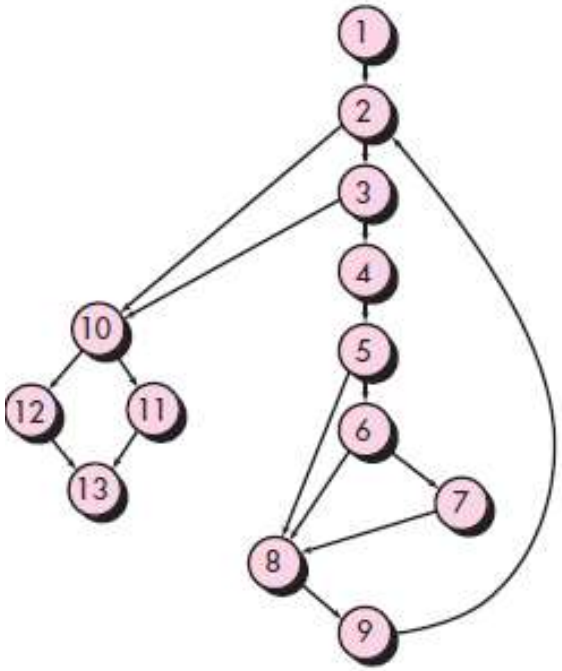
Step-1: Draw a corresponding flow graph:



White Box Testing – Basis Path Testing

Independent Program Paths

Example -6:



Flow Graph

Cyclomatic complexity:

$V(G)$ = 17 edges - 13 nodes + 2 = 6

$V(G)$ = 5 predicate nodes + 1 = 6 (2,3,5,6,10)

$V(G)$ = 6 regions

Path 1: 1-2-10-11-13

Path 2: 1-2-10-12-13

Path 3: 1-2-3-10-11-13

Path 4: 1-2-3-4-5-8-9-2-. . .

Path 5: 1-2-3-4-5-6-8-9-2-. . .

Path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable.

White Box Testing – Basis Path Testing

The complexity number and corresponding meaning of v (G):

Complexity Number	Meaning
1-10	Structured and well written code High Testability Cost and Effort is less
10-20	Complex Code Medium Testability Cost and effort is Medium
20-40	Very complex Code Low Testability Cost and Effort are high
>40	Not at all testable Very high Cost and Effort

White Box Testing – Control Structure Testing

- **Basic Path Testing**
- **Condition testing**
- **Data Flow Testing**
- **Loop Testing**

- **The above methods broaden testing coverage and improve the quality of white-box testing.**

White Box Testing – Control Structure Testing

1. Condition Testing:

- **Condition testing** is a test-case design method that **exercises the logical conditions** contained in a program module.
- A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator.
- A relational expression takes the form

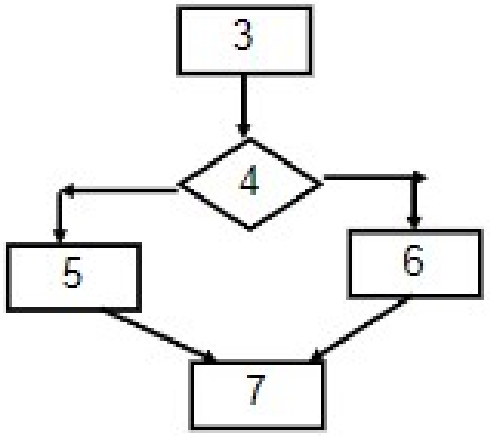
  E1 <relational-operator> E2

  where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, <=, >, >=, =, !=
- A **compound condition** is **composed of two or more simple conditions**, Boolean operators, and parentheses.
- Boolean operators are OR (|), AND (&), and NOT (¬).
- A condition **without relational expressions** is referred to as a **Boolean expression.**

White Box Testing – Control Structure Testing

2. Data Flow Testing:

1. Pseudo Code - Sample
2. int a, b //This is type defining not value
3. input (a, b)
4. if (a > b) then
5. Output (a, " a bigger than b")
6. else Output (b, " b is equal or greater than a")
7. end



- **DEF(a, 3)** – node 3 is a **defining node** of variable "a" --- value is assigned to "a"
- **USE(a, 4)** – node 4 is a **usage node** of variable "a"
- **USE(a, 5)** – node 5 is a **usage node** of variable "a"
- **USE(a,4)** is a **P-Use** node Decision making in a predicate (P-Use)
- **USE(a,5)** is **C-Use** node Computation and assignment (C-Use)
- **Path** that begins with DEF(a,3) and ends with USE(a,4) is a **definition-use (du-path) path of a**
- Path that begins with DEF(a,3) and ends with USE(a,5) is a **definition-use (du-path)** path of a
- Path that begins with DEF(a,3) and ends with USE(a,5) is a **definition-clear** path of a
- Path that begins with DEF(b,3) and ends with USE(b, 6) is a **definition-use** path of b
- Note: If the definition-use paths are chosen [last two examples above] of both variables a and b, then it is the same as executing the **decision-decision (dd) path or branch testing.**

White Box Testing – Control Structure Testing

2. Data Flow Testing:

```
1. read(x,y)
2. z = x+2
3. If(z<y)
4.      w = x+1
     else
5.     y= y+1
6. print(x,y,w,z);
```

Line	Def.	C-use	P-use
1	x,y		
2	z	x	
3			z,y
4	w	x	
5	y	y	
6		x,y,z,w	

White Box Testing – Control Structure Testing

2. Data Flow Testing:

- Selects test paths of a program according to the locations of **definitions** and **uses** of variables in the program
  - **From** the point where a variable, **v, is defined or assigned a value**
  - **To** the point where that variable**, v, is used**

Variable Define-Use Testing

- In define-use testing, testing is done on certain paths that a **variable is defined – to - its usage.**
- **Consider a data item, X:**
  - **Data Definitions (value assignment) of X:**
    1) Initialization → int X; (compiler initializes X to 0 or "trash")
    2) Input → Input X
    3) Some assignment → X = 3
  - **Data Usage (accessing the value) of X:**
    1) Computation and assignment **(C-Use)** → Z = X + 25
    2) Decision making in a predicate **(P-Use)** → If (X > 0) then

White Box Testing – Control Structure Testing

2. Data Flow Testing:

- **Defining node, DEF(v,n),** is a node, n, in the program graph where the specific variable, v, is defined or given its value (value assignment).
- **Usage node, USE(v,n),** is a node, n, in the program graph where the specific variable, v, is used.
- A **P-use node** is a usage node where the variable, v, is used as a **predicate (or for a branch-decision-making**).
- A **C-use node** is any usage node that is not P-used.
- A **Definition-Use path, du-path**, for a specific variable, v, is a path where DEF(v,x) and USE(v,y) are the initial and the end nodes (x and y nodes) of that path.
- A **Definition-Clear path** for a specific variable, v, is a Definition-Use path with DEF(v,x) and USE(v,y) such that there is **no other node in the path that is a defining node of variable, v**. (e.g. v does not get reassigned in the path.)

White Box Testing – Control Structure Testing

3. Loop Testing:

- Focuses exclusively on the validity of loop constructs.
- Four different classes of loops can be defined:
  - Simple loops
  - Concatenated loops
  - Nested loops
  - Unstructured loops

```
// Simple loop
for (int i = 1; i <= n; ++i)
{
  // codes
}
```

```
// concatenated loop
for (int i = 1; i <= 5; ++i)
{
  // codes
}
for (int j = 1; j <= 5; ++j) {
  // codes
}
```

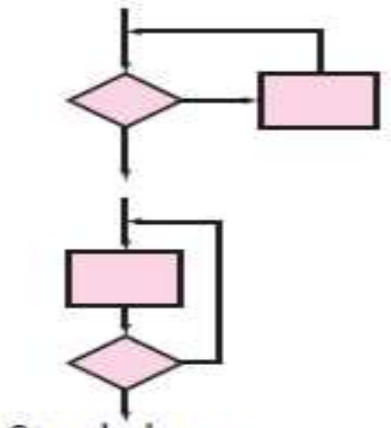
```
// concatenated loop
for (int i = 1; i <= 5; ++i)
{
  // codes
}
for (int j = 1; j <= i; ++j) {
  // codes
}
//j loop dependent on i
loop
```

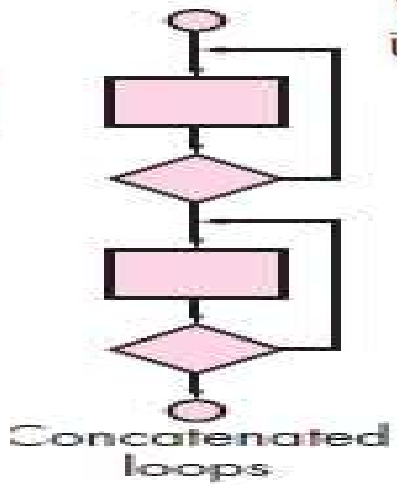
```
// outer loop
for (int i = 1; i <= 5; ++i)
{
  // codes
  // inner loop
     for(int j = 1; j <=n; ++j)
             {
                // codes
              }
}
```

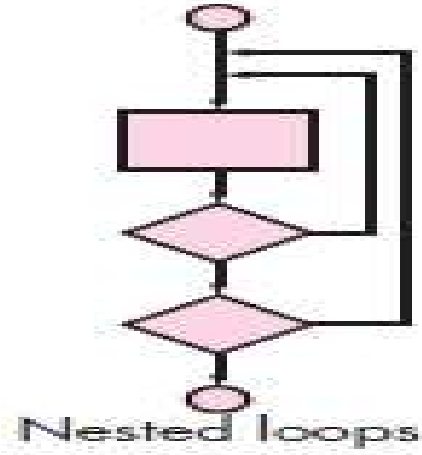
White Box Testing – Control Structure Testing

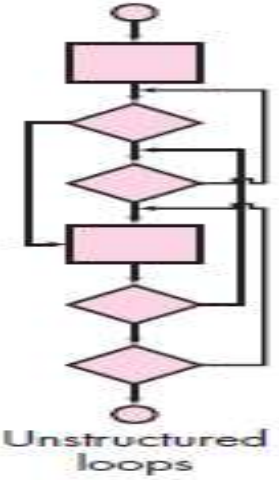
3. Loop Testing:

- Focuses exclusively on the validity of loop constructs.
- Four different classes of loops can be defined:
  - Simple loops
  - Concatenated loops
  - Nested loops
  - Unstructured loops





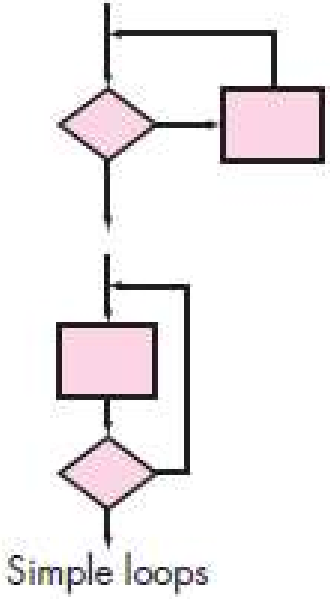




White Box Testing – Control Structure Testing

3. Loop Testing:

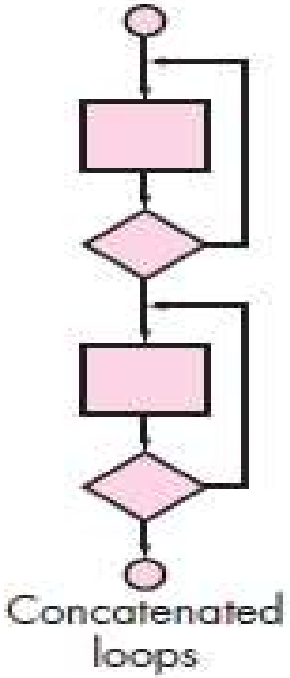
Simple loops:



- The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.
    1. Skip the loop entirely.
    2. Only one pass through the loop.
    3. Two passes through the loop.
    4. m passes through the loop where m < n.
    5. n -1, n, n + 1 passes through the loop.

White Box Testing – Control Structure Testing

3. Loop Testing:

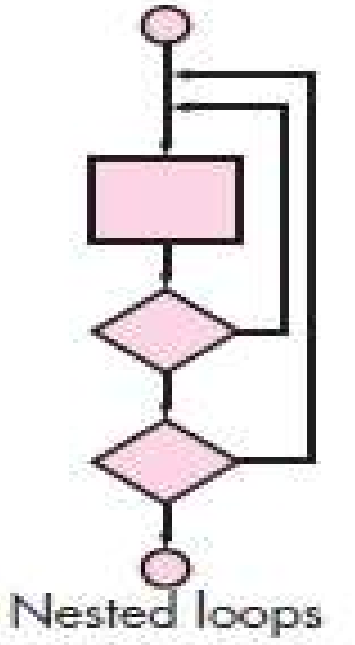


Concatenated loops:

- Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.
- However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are **not independent**. When the loops are not independent, the approach applied to nested loops is recommended.

White Box Testing – Control Structure Testing

3. Loop Testing:



Nested loops:

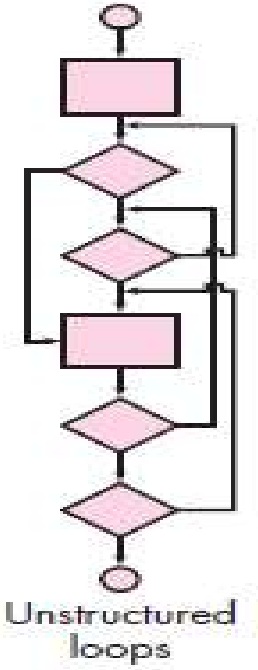
- The number of possible **tests would grow geometrically** as the **level of nesting increases.**

Approach

1. Start at the innermost loop, Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

White Box Testing – Control Structure Testing

3. Loop Testing:



Unstructured loops:

- Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

Black Box Testing

To buy a new car, and you have a few options in mind.
How do you evaluate which is better for you?
You will, most likely, take the car for a test drive and ask yourself:

- Is it comfortable to drive?
- Does it have enough power?
- Is the air-conditioning pleasant?
- Is the music system to your liking?
- Is the control panel easy to use?

You are not concerned about the internal machinery of the car.
Instead, you are interested in its functionality and usability.
This focus on the end-user is the principle behind black-box testing.

Black Box Testing

VIT-AP UNIVERSITY

Registration Form				
First name				
Last name				
Password				
Reenter Password				
E-Mail				
Mobile Number				
Gender	☐	Male	☐	Female
Date of Birth	DD/MM/YYYY			
Select Your State	▼			
Pin code				

Reset Submit

Black Box Testing

1. The pin code needs to be a number.
2. The phone number is 10 digit
3. If required fields are not filled or wrong data is inserted then form triggers error after submission of the form.

The test cases for the registration form designed as per fields

- Check the behavior of form by not filling up any data into the form.
- Check the form by not filling up the required fields.
- Check the behavior of form by adding random data in the text field.
- Check the by not filling up the first name text field but by filling up rest of the other fields.
- Check the form by filling other text fields except phone number text field.
- Check the form by filling other text fields except email text field etc..,

Black Box Testing

Testing Name Fields

- Check the Name text fields with special characters
- Check by adding numbers instead of string in the name text fields

Testing Password Fields

- Check the Password text field by adding no data.
- Check the Password text field by adding numbers into it.
- Check the Password text field by adding data more than the field limit.
- Check the Reenter Password text field by adding no data
- Check the Reenter Password text field by adding mismatch data

Testing other Fields

- Check the behavior of form by not selecting the state from the drop-down menu of the country field.
- Check the pincode / mobile number text field with string instead of numbers.
- Check the pincode / mobile number text field with numbers shorter than required.
- Check the pincode / mobile number field with numbers larger than required

Black Box Testing

VIT-AP UNIVERSITY

Testing Email Field

- Check the Email text field that has Email address without @ symbol.
- Check the Email text field that has random string instead of real email.
- Check the Email text field that has @ symbol written in words.
- Check the Email text field that has missing dot in the email address.
- Check the Email text field as “name@gmail”
- Check the Email text field as “@gmail”
- Check the Email text field as “name@gmail..com”
- Check the Email text field as “name@192.168.1.1.0”
- Check the Email text field as “name.. @gmail.com”

Black Box Testing

- A software testing method in which the functionalities of software are tested **without having knowledge of internal code structure, implementation details and internal paths.**
- Also called **Behavioral testing,** **focuses** on the **functional requirements and specifications** of the software.
- Techniques **enable to derive sets of input conditions** that will **fully exercise all functional requirements** for a software**.**
- It 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

VIT-AP UNIVERSITY

- 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 tends to be applied during later stages of testing, because black-box testing purposely disregards control structure, attention is focused on the information domain.

Black Box Testing - Graph-Based Testing Methods:

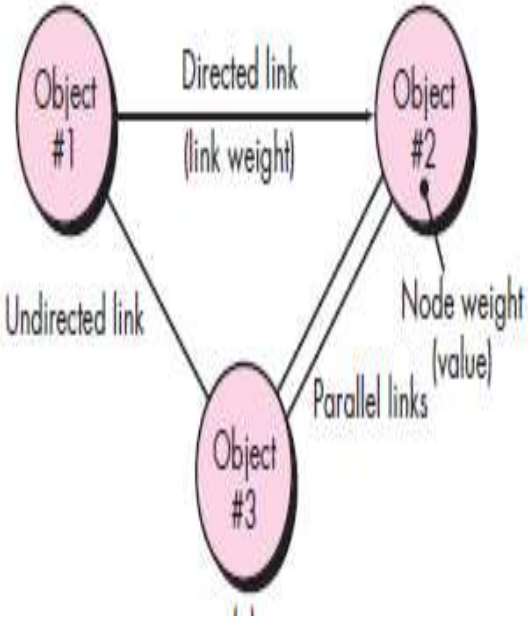
- The **first step** in black-box testing is to **understand the objects** that are modeled in software and the **relationships that connect these objects**.

- The **next step** is to **define a series of tests** that verify **"all objects have the expected relationship to one another"**

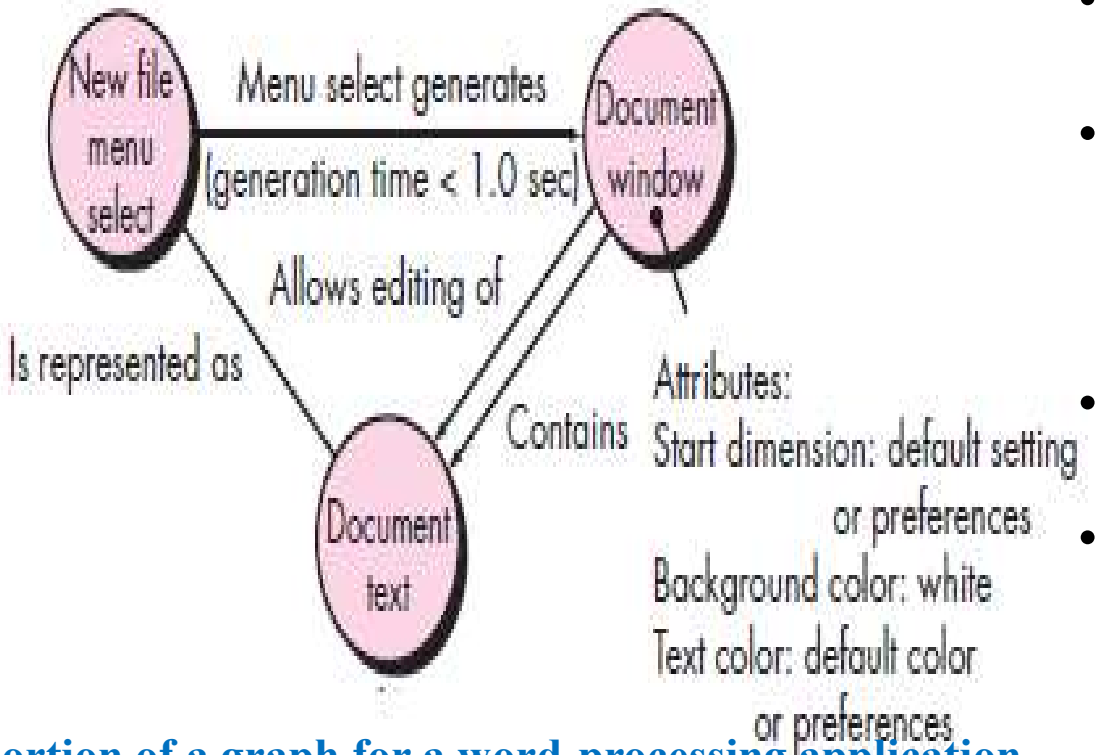
- Software testing **begins** by
  - Creating a graph of important objects and their relationships
  - Devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

Black Box Testing - Graph-Based Testing Methods:

- **Nodes (objects)** are represented as **circles** connected by links.
- **Links** represent the **relationships** between objects
- **Node weights** describe the properties of a node (e.g., a specific data value or state behavior), and
- **Link weights** that describe **characteristic of a link**, which can be
  - A **directed link** (represented by an arrow) indicates that a relationship moves in only one direction.
  - A **bidirectional link**, also called a symmetric link, implies that the relationship applies in both directions.
  - **Parallel links** are used when a number of different relationships are established between graph nodes.



Black Box Testing - Graph-Based Testing Methods:



Portion of a graph for a word-processing application

Object #1 **newFile** (menu sélection),
Object #2 **documentWindow**
Object #3 **documentText**

- A menu select on **newFile** generates a document window.
- The node weight of **documentWindow** provides a list of the window attributes that are to be expected when the window is generated.
- The link weight indicates that the window must be generated in less than 1.0 second.
- An undirected link establishes a symmetric relationship between the **newFile** menu selection and **documentText,** and parallel links indicate relationships between **documentWindow** and **documentText.**

Black Box Testing - Graph-Based Testing Methods:

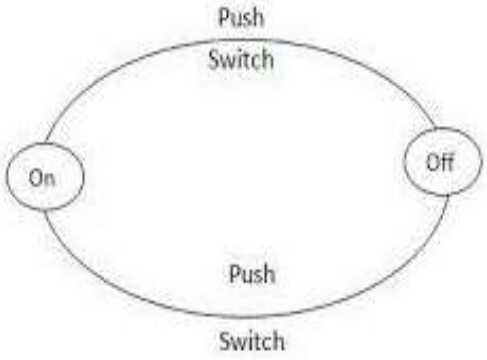
Behavioral testing methods that can make use of graphs:

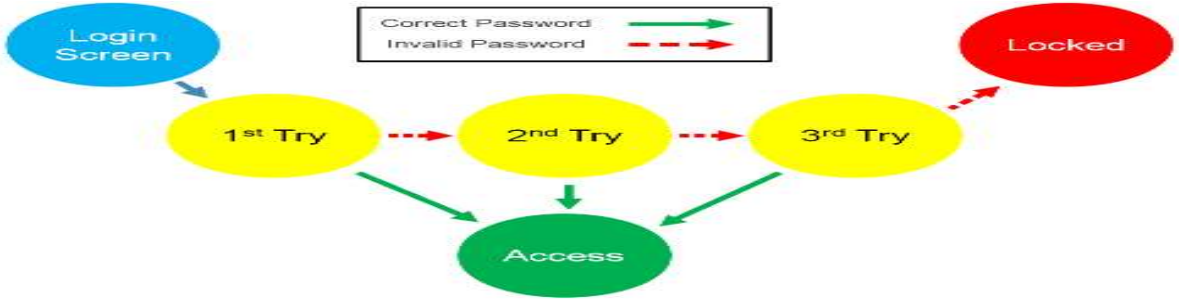
1. Transaction flow modelling:

- The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and
- The links represent the logical connection between steps (e.g., flight_Information_Input is followed by validation_Availability_Processing).
- The **data flow diagram** can be used to assist in creating graphs of this type.

2. Finite state modelling: State Transition Diagram and State Transition Table

- The nodes represent different user-observable states of the software
- The links represent the transitions that occur to move from state to state
- The state diagram can be used to assist in creating graphs of this type.





An ATM system function where if the user enters the invalid password three times the account will be locked.

Black Box Testing - Graph-Based Testing Methods:

3. Data flow modelling:

- The nodes are data objects
- The links transformations that occur to translate one data object into another.

4. Timing modelling:

- The nodes are program objects,
- The links are the sequential connections between those objects.
- Link weights are used to specify the required execution times as the program executes.

Black Box Testing -Equivalence Partitioning

- **Divides** the input values into **different groups or classes.**
  - This is done on the basis of the output which will be coming as an outcome.
- It saves the effort and time of giving different inputs. Input for each class is sufficient.
- Instead, give one value to the group or class to test the outcome for that group or class.
- Helps in **improving the test coverage** and in turn **reducing the rework**.

Example: If a student scores

Above 75%	- First class with Distinction.
Between 60% to 75%	- First Class
Between 50% to 60%	- Second Class.
Between 40% to 50%	- Pass class,
Else	- Fail.

- Test cases are formed and it is made sure that all possibilities are covered.
- Hence testing with any values in this set is sufficient.

Black Box Testing -Equivalence Partitioning

Guidelines for defining Equivalence Partitioning classes:

1. If an input condition **specifies a range**, one valid and two invalid equivalence classes are defined.
   Example: 100-500, Valid: 145, Invalid: 80, 501
2. If an input condition requires a **specific value**, one valid and two invalid equivalence classes are defined.
   Example: OTP – 6 digits [Mobile Number – 10 digits]
   Valid – 6 digits Invalid: No.of Digits >= 7 and No.of Digits <=5
3. If an input condition specifies a **member of a set**, one valid and one invalid equivalence class are defined.
4. If an input condition is **Boolean**, one valid and one invalid class are defined.
   Example: Male / Female

Black Box Testing - Boundary Value Analysis (BVA)

- A greater number of errors occurs at the boundaries of the input domain rather than in the "center."
- Boundary Value Analysis (BVA) has been developed as a testing technique that leads to a selection of **test cases that exercise bounding values**.

Boundary Value Analysis

Age | **Enter Age** | **Accepts value between 18 to 65**

Invalid Test Case (Min Value -1)	Valid Test Case (Min, +Min, Max, -Max)	Invalid Test Case (Max Value +1)
17	18,19,25,64,65	66

Black Box Testing - Boundary Value Analysis (BVA)

VIT-AP UNIVERSITY

Guidelines for defining BVA:

1. If an input condition **specifies a range** bounded by values a and b, test cases should be designed with values a and b and **just above and just below a and b**.
2. If an input condition specifies a **number of values**, test cases should be developed that exercise the minimum and maximum numbers. **Values just above and below minimum and maximum are also tested.**
3. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Black Box Testing - Boundary Value Analysis (BVA)

Example:

When thinking about a buying situation,

- If you spend Rs. 500, you'll get a 5% discount.
- If you spend Rs. 1000, you'll get a 7% discount.
- If you spend Rs. 1500 or more, you'll get a 10% discount.

- It is feasible to split inputs into four divisions using the **Equivalence partitioning** approach of this testing, with amounts less than 0, 0 – 500, 501 – 1000, 1001 – 1500, and so on.
- **Equivalence partitioning** testing approach will not take into account variables such as the maximum shopping limit or product specifications.
- The boundary values will be 0, 500, 501, 1000, 1001, and 1500 when boundary values are added to the partitions.
- The lowest and higher values are normally evaluated using the **BVA approach**, therefore numbers like -1, 1 and 499 will be included. Such values will aid in the explanation of software input value behavior.

Black Box Testing - Error Guessing

- This is an **Experience-Based Testing**.
- In this technique, the tester can use their experience about the **application behavior and functionalities to guess the error-prone areas.**
- Many defects can be found using error guessing where most of the developers usually make mistakes.
- **Few common mistakes that developers usually forget to handle:**
  - Divide by zero.
  - Handling null values in text fields.
  - Accepting the Submit button without any value.
  - File upload without attachment.
  - File upload with less than or more than the limit size.

Black Box Testing - Decision Table Testing

- Captures different input combinations and their expected results in a tabular form and design test cases based on this table.
- Decision table is carved to prepare a set of test cases.
- **Example:**

  An XYZ bank provides an interest rate for the Male senior citizen as 10% and 9% for the rest of the people.

Decision Table / Cause-Effect

Decision Table	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
C1 - Male	F	F	T	T
C2 - Senior Citizen	F	T	F	T
Actions				
A1 - Interest Rate 10%				X
A2 - Interest Rate 9%	X	X	X	

- C1 has two values as true and false, C2 also has two values as true and false.
- The total number of possible combinations would then be four.
- Derive test cases using a decision table.

Black Box Testing - Decision Table Testing

Example:
A job portal where users can upload their resume such that:
The file has to be a PDF document.
Its size must not exceed 1 MB.

Format	Size	Output	
Case 1	PDF	≤ 1MB	Success
Case 2	PDF	> 1MB	Error: file size
Case 3	Not PDF	≤ 1MB	Error: format
Case 4	Not PDF	> 1MB	Error: format and file size

- If there is another constraint that the number of pages cannot exceed 5, there will be additional rows in the table.
- Once table is ready, write test cases corresponding to every row.

Black Box Testing - Orthogonal Array Testing (OAT)

A Web page has three distinct sections (Top, Middle, Bottom) that can be individually shown or hidden from a user

In Conventional testing technique, required test cases are 6 (2 X 3 = 6)

Test Cases	Scenarios	Values to be tested
Test #1	HIDDEN	Top
Test #2	SHOWN	Top
Test #3	HIDDEN	Bottom
Test #4	SHOWN	Bottom
Test #5	HIDDEN	Middle
Test #6	SHOWN	Middle

Black Box Testing - Orthogonal Array Testing (OAT)

- In the **conventional method**, test suites include test cases that have been derived from all combination of input values and pre-conditions.
  - As a result, **n** number of test cases has to be covered.
  - But in **a real scenario**, the testers won't have the leisure to execute all the test cases to uncover the defects as there are other processes such as documentation, suggestions, and feedback from the customer that has to be taken into account while in the testing phase.
- **Test Case Optimization :** The test managers wanted to optimize the number and quality of the test cases to ensure maximum Test coverage with minimum effort.
- **Orthogonal Array Testing (OAT):**
  - Uses orthogonal arrays to create test cases.
  - It is a statistical testing approach **useful when system to be tested has huge data inputs**.
  - Helps to maximize test coverage by pairing and combining the inputs and testing the system with comparatively less number of test cases for time saving.

Black Box Testing - Orthogonal Array Testing (OAT)

The formula to calculate OAT

$$L_{Runs}(Levels^{Factors})$$

Term	Description
Runs	It is the number of rows which represents the number of test conditions to be performed.
Factors	It is the number of columns which represents in the number of variable to be tested
Levels	It represents the number of values for a Factor

- The goal is to minimize the number of rows as much as possible.

Black Box Testing - Orthogonal Array Testing (OAT)

The Orthogonal Array Testing technique has the following steps:

1. Decide the number of variables that will be tested for interaction. Map these variables to the factors of the array.
2. Decide the maximum number of values that each independent variable will have. Map these values to the levels of the array.
3. Find a suitable orthogonal array with the smallest number of runs. The number of runs can be derived from various websites. One such website is listed here.
4. Map the factors and levels onto the array.
5. Translate them into the suitable Test Cases
6. Look out for the leftover or special Test Cases (if any)

Black Box Testing - Orthogonal Array Testing (OAT)

A Web page has three distinct sections (Top, Middle, Bottom) that can be individually shown or hidden from a user

- **Step 1:** Determine the number of independent variables. There are three independent variables (sections on the page) = **3 Factors.**
- **Step 2:** Determine the maximum number of values for each variable. There are two values (hidden and visible) = **2 Levels.**
- **Step 3:** Determine the Orthogonal Array with 3 Factors and 2 Levels.
- Referring to the link we have derived the number of rows required i.e. **4 Rows.**

The orthogonal array follows the pattern $L_{Runs}(Levels^{Factors})$.
Hence, the Orthogonal Array will be $\mathbf{L_4(2^3)}$.

Black Box Testing - Orthogonal Array Testing (OAT)

OAT Testing need 4 Test cases as shown below:

Test Cases	TOP	Middle	Bottom
Test #1	Hidden	Hidden	Hidden
Test #2	Hidden	Visible	Visible
Test #3	Visible	Hidden	Visible
Test #4	Visible	Visible	Hidden

Hence while executing such test conditions, a tester will put the conditions as follows:

- Display the Homepage and hide all sections.
- Display the Homepage and show all Sections except Section 1.
- Display the Homepage and show all Sections except Section 2.
- Display the Homepage and show all Sections except Section 3.