Osnovi softverskog inženjerstva

P-10: Verifikacija i validacija

2024

Topics covered

- Verification & Validation
- Reviews
- Software testing
- Software testing strategies
  - User testing / Integration testing / Validation testing / System testing
- Test-driven development
- Software testing techniques
  - White-box testing / Black-box testing

Verification & Validation

Verification & Validation

- **Verification:**
  "Are we building the product right".
  - The software should conform to its specification.
- **Validation:**
  "Are we building the right product".
  - The software should do what the user really requires.

V&V confidence

- **Aim of V&V is to establish confidence that the system is 'fit for purpose'.**
- **Depends on system's purpose, user expectations and marketing environment**
  - **Software purpose**
    - The level of confidence depends on how critical the software is to an organisation.
  - **User expectations**
    - Users may have low expectations of certain kinds of software.
  - **Marketing environment**
    - Getting a product to market early may be more important than finding defects in the program.

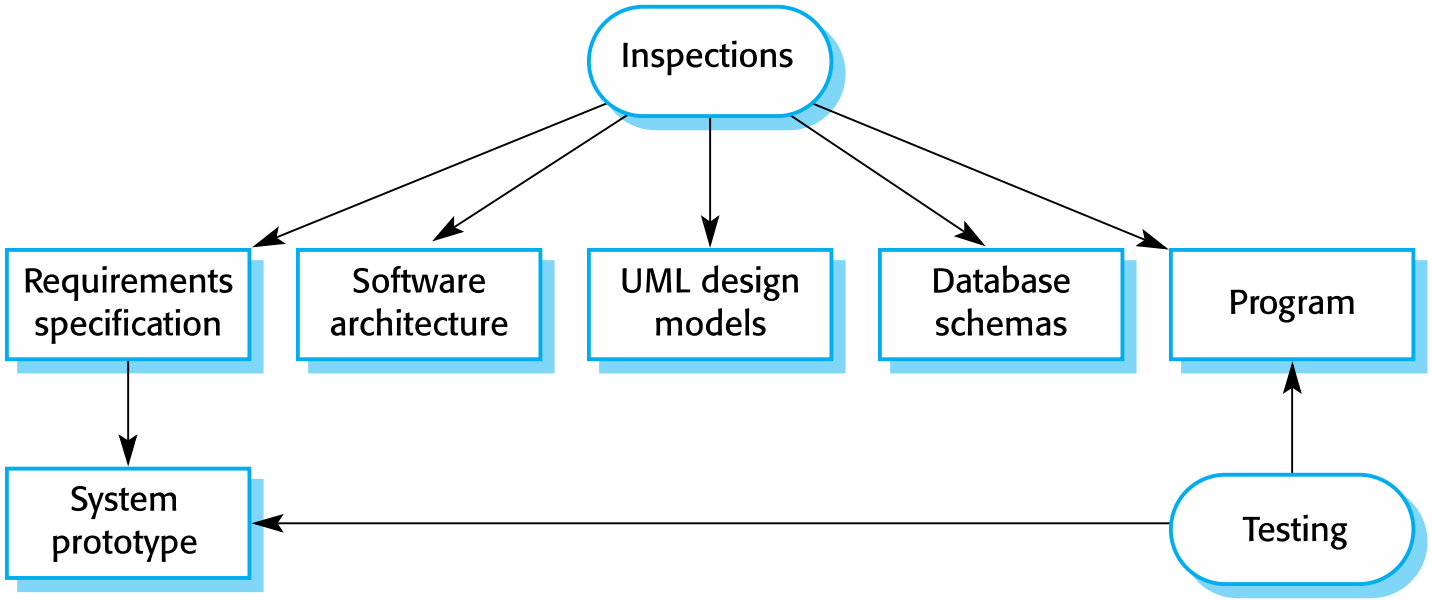
V&V approaches

Static approaches

- **Concerned with analysis of the static system representation to discover problems (static verification)**
- **Reviews/inspections**
- May be supplement by tool-based document and code analysis.

Dynamic approaches

- **Concerned with exercising and observing product behaviour (dynamic verification)**
- **Software testing**
- The system is executed with test data and its operational behaviour is observed.



Strategic approach to V&V

- To achieve effective V&V:
  - We 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.

Reviews

What Are Reviews?

- a meeting conducted by technical people for technical people
- a technical assessment of a work product created during the software engineering process
- a software quality assurance mechanism
- a training ground

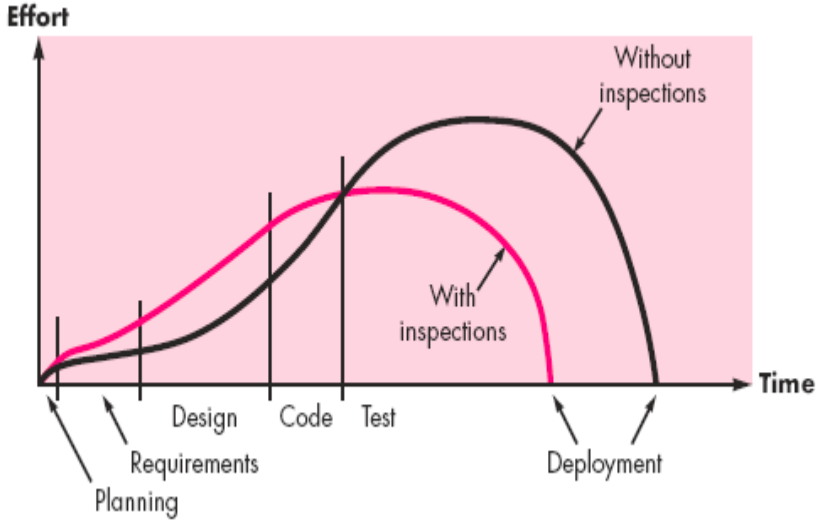
What Reviews Are Not?

- A project summary or progress assessment
- A meeting intended solely to transfer information
- A mechanism for political or personal reprisal!

What Do We Look For?

- **Errors and defects**
  - **Error** — a quality problem found before the software is released to end users
  - **Defect** — a quality problem found only after the software has been released to end-users
- We make this distinction because errors and defects have very different economic, business, psychological, and human impact (However, this temporal distinction made between errors and defects is not mainstream thinking)

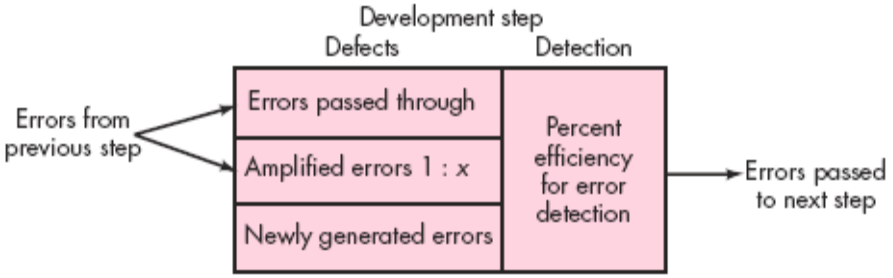
Cost Effectiveness of Reviews



- The effort expended when reviews are used does increase early in the development of a software increment, but this early investment for reviews pays dividends because testing and corrective effort is reduced.
- **The deployment date for development with reviews is sooner than the deployment date without reviews.**
- **Reviews don't take time, they save it!**

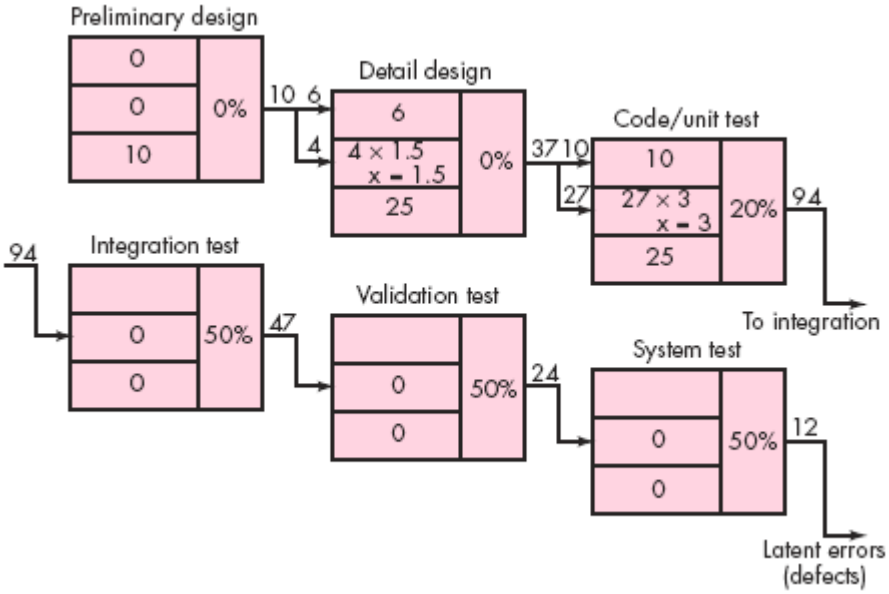
Defect Amplification

- A defect amplification model can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process



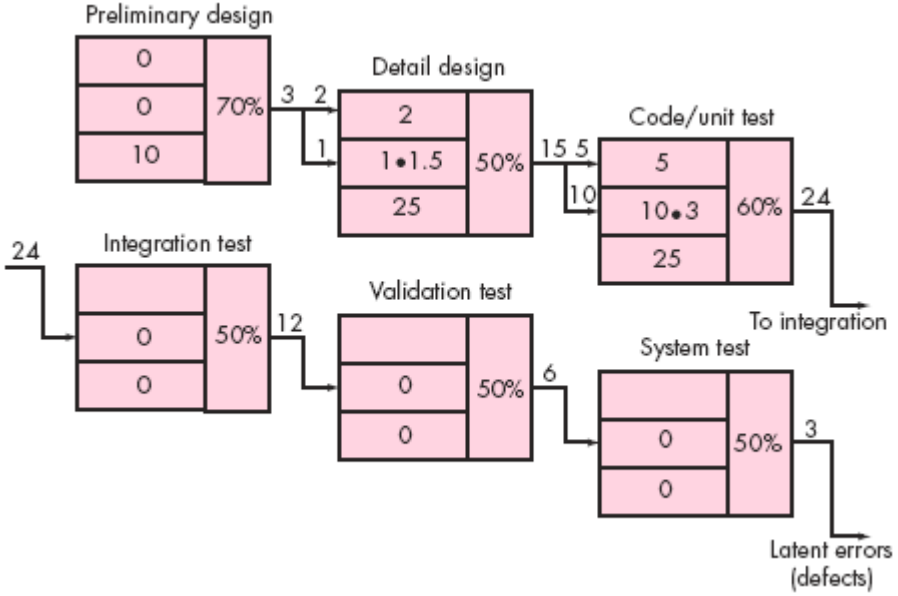
Defect Amplification Examples

No reviews conducted



- a software process that does NOT include reviews:
  - yields 94 errors at the beginning of testing
  - releases 12 latent defects to the field

Reviews conducted



- a software process that does include reviews:
  - yields 24 errors at the beginning of testing
  - releases 3 latent defects to the field

A cost analysis indicates that the process with NO reviews costs approximately 3-4 times more than the process with reviews, taking the cost of correcting the latent defects into account

Informal Reviews

- **Informal reviews include**:
  - a **simple desk check** of a software engineering work product with a colleague
  - a **casual meeting** (involving more than 2 people) for the purpose of reviewing a work product, or
  - the **review-oriented aspects of pair programming**
- **pair programming** encourages **continuous review** as a work product (design or code) is created.
  - The benefit is immediate discovery of errors and better work product quality as a consequence.

Formal Technical Reviews

- **The objectives of an FTR are**:
  - to **uncover errors** in function, logic, or implementation for any representation of the software
  - to **verify that the software** under review **meets its requirements**
  - to **ensure that the software has been represented according to predefined standards**
  - to **achieve software that is developed in a uniform manner**
  - to **make projects more manageable**
- **The FTR is actually a class of reviews that includes walkthroughs and inspections**.

The FTR Meeting

- **3-5 people (**typically) should be **involved in the review**.
- **Advance preparation** should occur but should require **no more than 2 hours** of work **for each participant**.
- **The duration of the review meeting should be less than 2 hours.**
- **Focus is on a work product**, e.g.
  - **a portion of a requirements model,**
  - **a detailed component design,**
  - **source code for a component**

The FTR Players

- **Producer —** the individual who has developed the work product
  - informs the project leader that the work product is complete and that a review is required
- **Review leader —** evaluates the product for readiness, generates copies of product materials, and distributes them to 2-3 reviewers for advance preparation.
- **Reviewer(s) —** expected to spend 1-2 hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- **Recorder —** reviewer who records (in writing) all important issues raised during the review.

Conducting the FTR

- Review the product, not the producer!
- Set an agenda and maintain it!
- Limit debate and rebuttal!
- Enunciate problem areas, but don't attempt to solve every problem noted!
- Take written notes!
- Limit the number of participants and insist upon advance preparation!
- Develop a checklist for each product that is likely to be reviewed!
- Allocate resources and schedule time for FTRs.
- Conduct meaningful training for all reviewers!
- Review your early reviews!

Sample-driven Review (SDR)

- Ideally, every product would undergo a FTR, but in reality – resources are limited and time is short – **reviews are often skipped**
- **SDR approach:**
  - **firstly, only product samples are inspected to determine the most error prone elements.**
  - **full FTR resources are then focused only on those products that are likely (based on the sample data)**
- **SDR process:**
  - Inspect a fraction (a_i) of each software work product *i*.
  - Record the number of faults (f_i) found within a_i.
  - Estimate the total number of faults within work product i by multiplying f_i *$1/a_i$.
  - Sort the products in descending order according to the estimated number of faults in each.
  - Focus available review resources on those products having the highest estimated number of faults.

Software testing

Software testing

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.
- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- Testing has a dynamic nature – it includes program execution (using some artificial data).
- We check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- Testing can reveal the presence of errors, but CAN NOT their absence.
- Testing is part of a more general V&V process, which also includes static validation techniques (reviews).

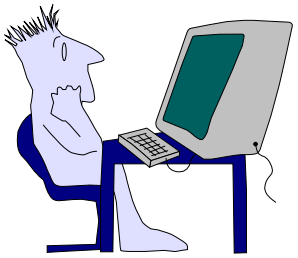
Software testing goals

- To demonstrate to the developer and the customer that the software meets its requirements.
  - For custom software, this means that there should be at least one test for every requirement in the requirements document.
  - For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

Validation and defect testing

- **The first goal leads to validation testing**
  - To demonstrate to the developer and the system customer that the software meets its requirements
  - A successful test shows that the system operates as intended.
- **The second goal leads to defect testing**
  - To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Who Tests the Software?



developer

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

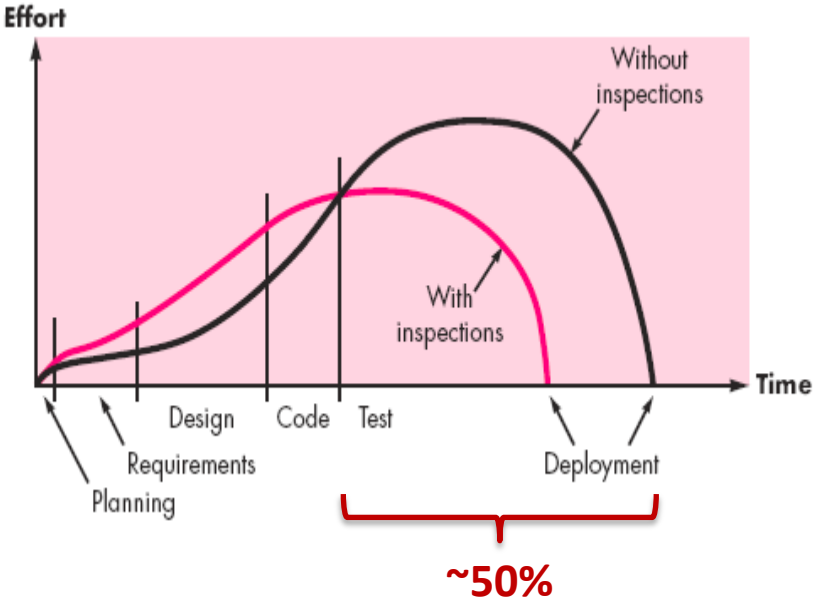
independent tester

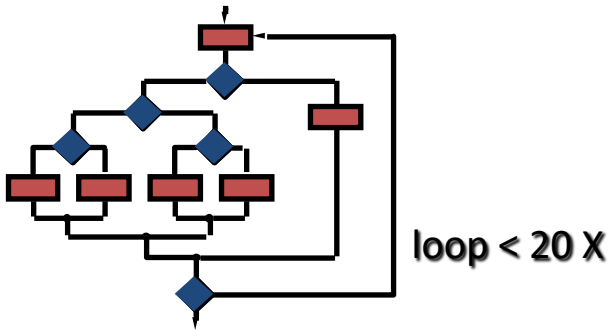
Must learn about the system,
but will attempt to break it and,
is driven by quality

(In)dependent testers

- Testing should aim at "breaking" the software
- **Common misconceptions:**
  - The developer of software should do no testing at all
  - The software should be given to a secret team of testers who will test it unmercifully
  - The testers get involved with the project only when the testing steps are about to begin
- **Reality: Independent test group**
  - Removes the inherent problems associated with letting the builder test the software that has been built
  - Removes the conflict of interest that may otherwise be present
  - Works closely with the software developer during analysis and design to ensure that thorough testing occurs

Testing efforts





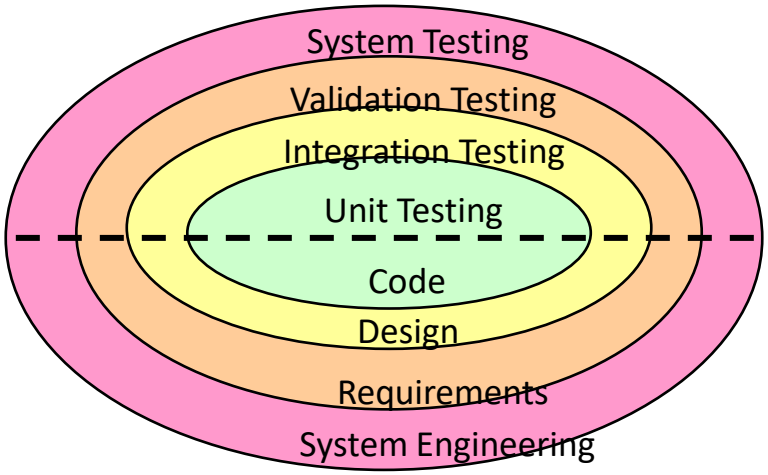
There are 10^{14} possible paths!
If we execute one test per millisecond, it would take 3,170 years to test this program!!

When is testing complete?

- **There is no definitive answer to this question!**
- **Every time a user executes the software, the program is being tested**
- **Sadly, testing usually stops when a project is running out of time, money, or both**
- **One approach is to divide the test results into various severity levels. Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated**

Levels of Testing

- **Unit testing**
  - Concentrates on each component/function of the software as implemented in the source code
- **Integration testing**
  - Focuses on the design and construction of the software architecture
- **Validation testing**
  - Requirements are validated against the constructed software
- **System testing**
  - The software and other system elements are tested as a whole



Sequence of four steps

1. Unit testing

- **Concentrates on each component/function of the software as implemented in the source code**
- Unit testing makes **heavy use of testing techniques that exercise specific paths in a component's control structure** to ensure complete coverage and maximum error detection.

2. Integration testing

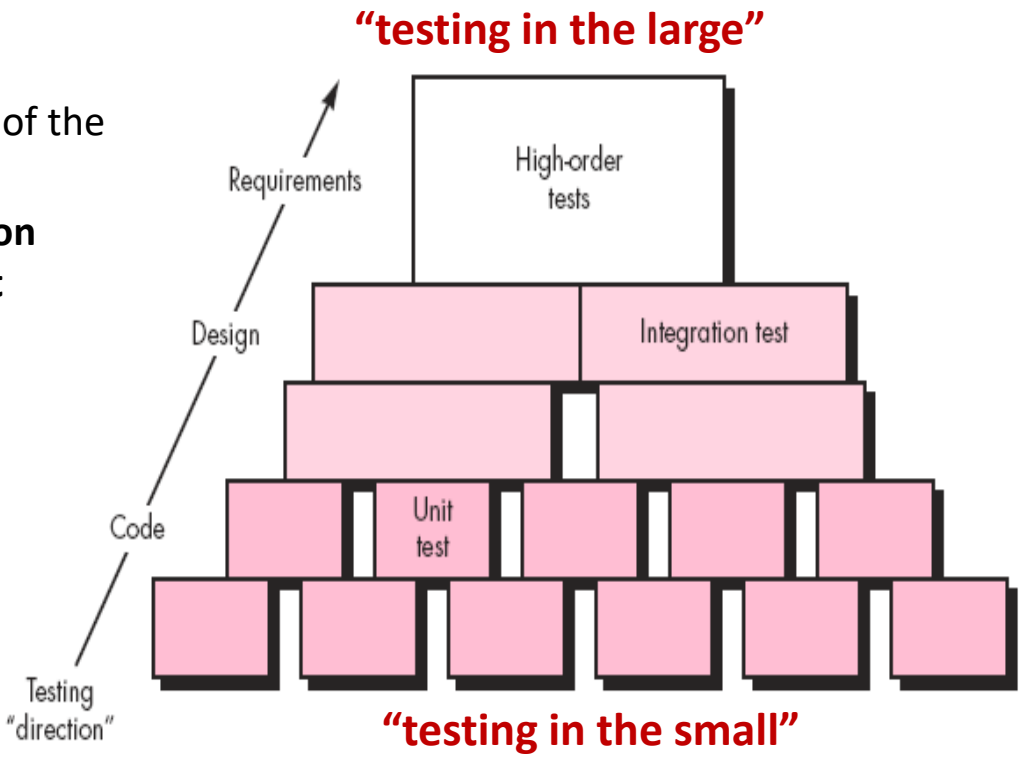
- Focuses on the design and construction of the software architecture
- **Test case design techniques that focus on inputs and outputs are more prevalent**

3. Validation testing

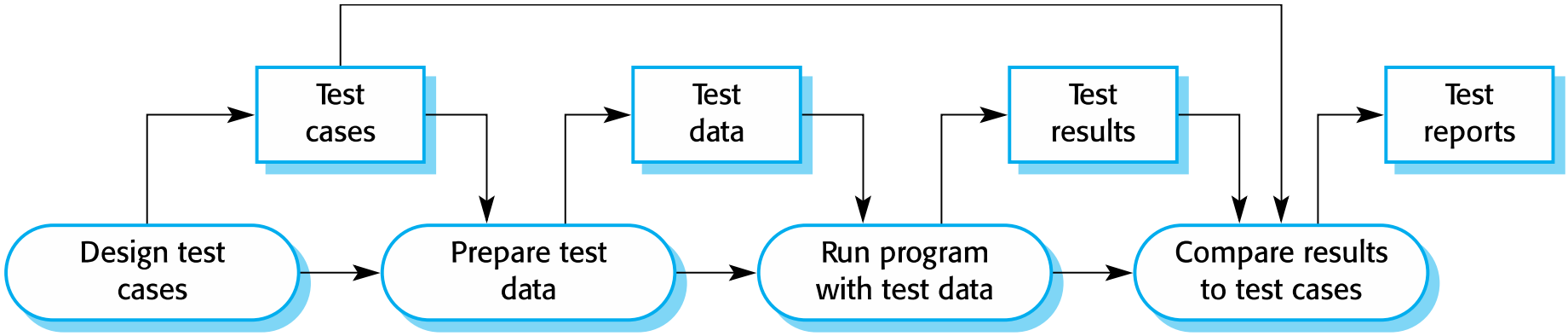
- Requirements are validated against the constructed software
- It provides final assurance that software meets all informational, functional, behavioral, and performance requirements

4. System testing

- The software and other system elements are tested as a whole
- It verifies that all elements mesh properly and that overall system function/performance is achieved.



A model of the software testing process



Test Report example

Project Name:	**Google Email**
Module Name:	**Login**
Reference Document:	**If any**
Created by:	**Rajkumar**
Date of creation:	**DD-MMM-YY**
Date of review:	**DD-MMM-YY**

STM

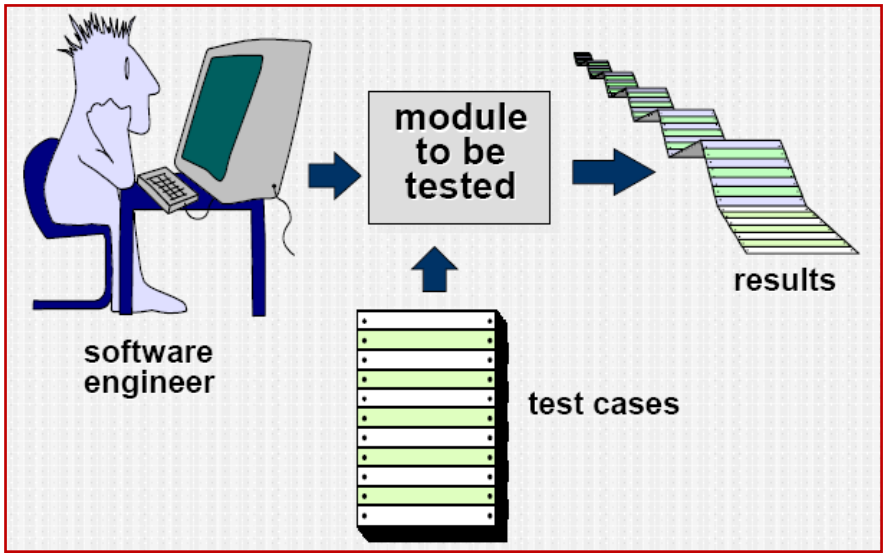
www.SoftwareTestingMaterial.com

TEST CASE ID	TEST SCENARIO	TEST CASE	PRE-CONDITION	TEST STEPS	TEST DATA	EXPECTED RESULT	POST CONDITION	ACTUAL RESULT	STATUS (PASS/ FAIL)
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Valid User Name>	Successful login	Gmail inbox is shown		
				2. Enter Password	<Valid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Valid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Invalid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Invalid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Valid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Invalid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Invalid Password>				
				3. Click "Login" button					

Software testing strategies

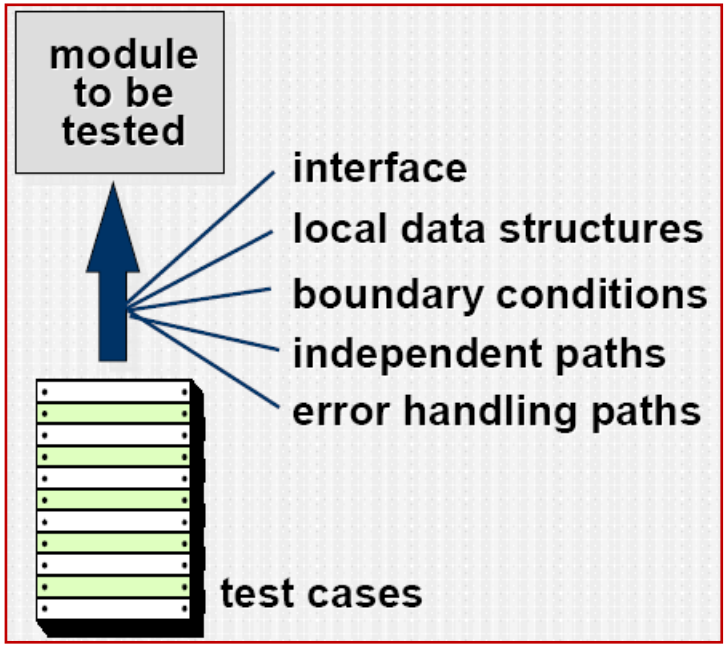
Unit testing

- **Unit testing is the process of testing individual components in isolation.**
- **It is a defect testing process.**
- **Units may be:**
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality.
- **Concentrates on the internal processing logic and data structures**
- **Concentrates on critical modules and those with high cyclomatic complexity when testing resources are limited**



Targets for Unit Test Cases

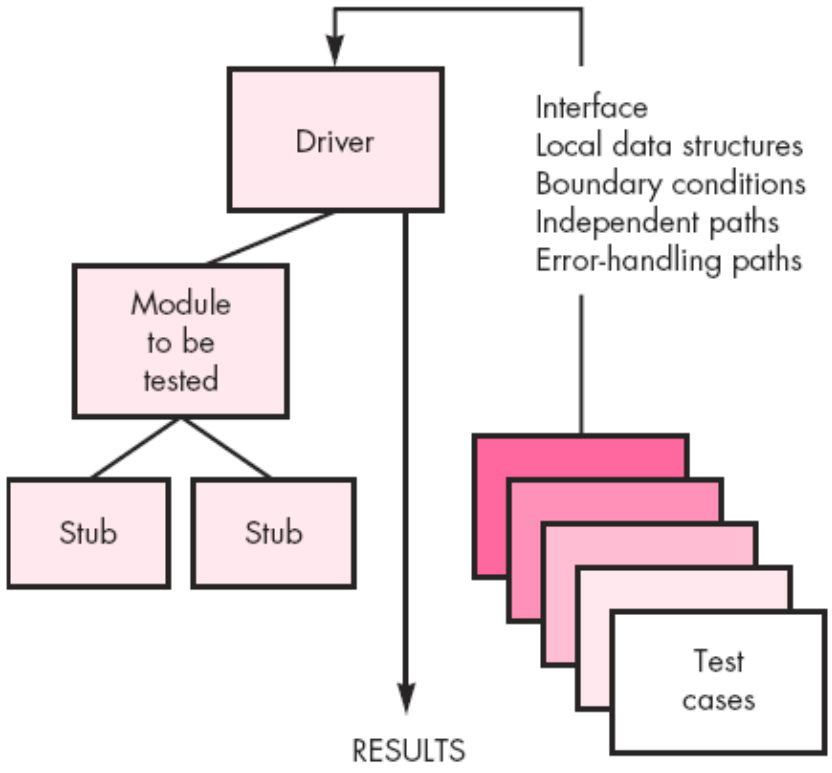
- **Module interface**
  - Ensure that information flows properly into and out of the module
- **Local data structures**
  - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- **Boundary conditions**
  - Ensure that the module operates properly at boundary values established to limit or restrict processing
- **Independent paths (basis paths)**
  - Paths are exercised to ensure that all statements in a module have been executed at least once
- **Error handling paths**
  - Ensure that the algorithms respond correctly to specific error conditions



Some unit testing techniques will be presented later

Unit Test Procedure

- Unit testing is normally considered as an adjunct to the coding step.
- The design of unit tests can occur before coding begins or after source code has been generated.
- A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier.
- Each test case should be coupled with a set of expected results.
- **Driver – A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results**
- **Stubs – Serve to replace modules that are called by the component to be tested**
  - A stub uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing



Drivers and stubs both represent overhead

Both must be written but don't constitute part of the installed software product

Regression testing

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- **Regression testing re-executes a small subset of tests that have already been conducted**
- **Helps to ensure that changes do not introduce unintended behavior or additional errors**
- **May be done manually or through the use of automated capture/playback tools**
- **Regression test suite 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 actual software components that have been changed

Automated unit testing

- Whenever possible, **unit testing should be automated** so that tests are run and checked without manual intervention.
- **In automated unit testing, you make use of a test automation framework** (such as **JUnit**) to write and run your program tests.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.

Integration testing

- Defined as a systematic technique for constructing the software architecture
- At the same time integration is occurring, conduct tests to uncover errors associated with interfaces
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two approaches
  - Non-incremental Integration testing
    (the "Big Bang" approach)
  - Incremental Integration testing
    - Top-down integration
    - Bottom-up integration
    - Sandwich integration

The "Big Bang" approach

- All components are combined in advance
- The entire program is tested as a whole
- Chaos results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

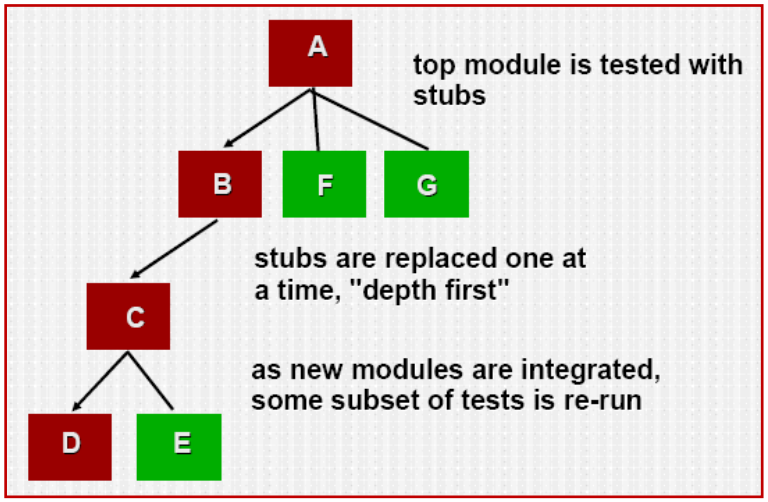
Incremental Integration Strategies

- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

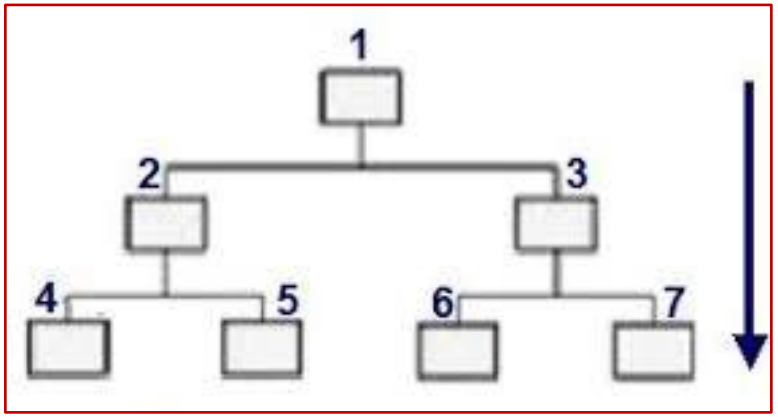
Top-down integration strategy

- **Modules are integrated by moving downward through the control hierarchy, beginning with the main module**
- **Subordinate modules are incorporated in either a depth-first or breadth-first fashion**
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated
- **Advantages**
  - This approach verifies major control or decision points early in the test process
- **Disadvantages**
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration process

DF integration strategy

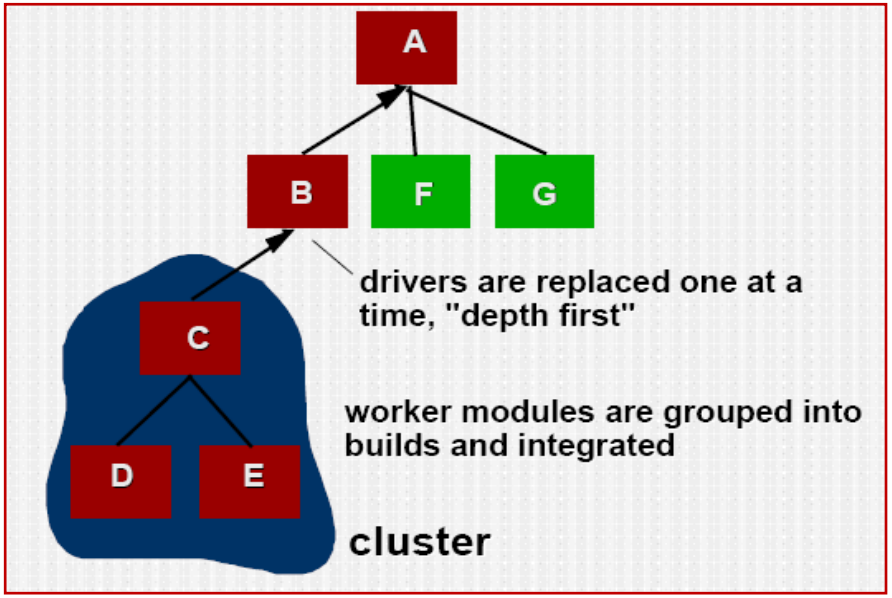


BF integration strategy



Bottom-up integration strategy

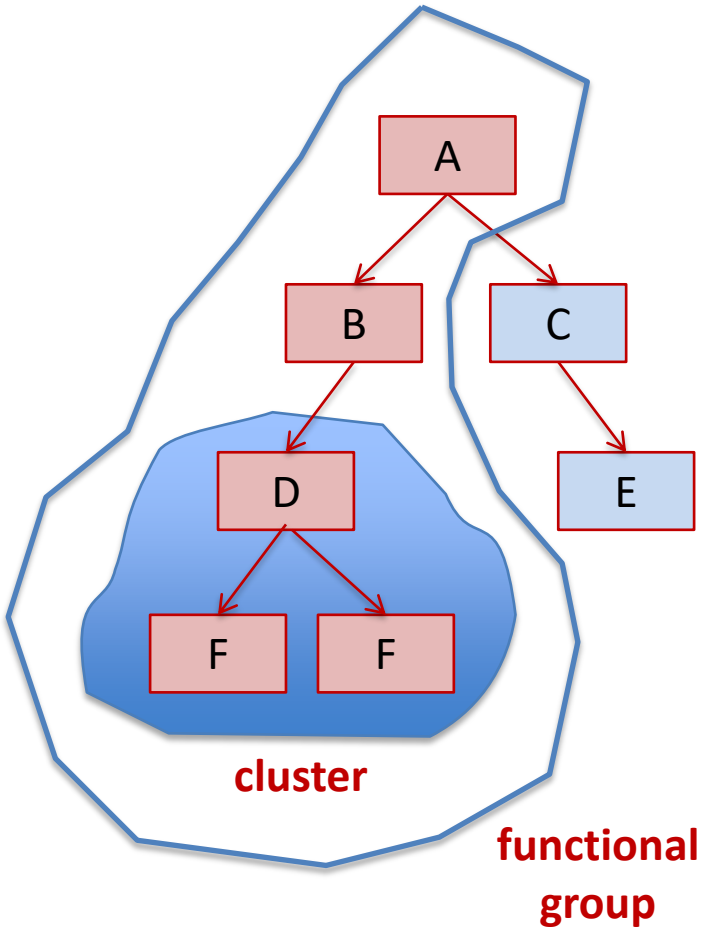
- **Integration and testing starts with the most atomic modules in the control hierarchy**
- **Advantages**
  - This approach verifies low-level data processing early in the testing process
  - Need for stubs is eliminated
- **Disadvantages**
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available



1. **Low-level components are combined into clusters** that perform a specific software subfunction.
2. **A driver is written to coordinate test case input and output.**
3. **The cluster is tested**.
4. **Drivers are removed** and **clusters are combined moving upward in the program structure**.

Sandwich integration strategy

- **Combines both top-down and bottom-up approaches**
- Occurs both at the highest level modules and also at the lowest level modules
- **Proceeds using functional groups of modules, with each group completed before the next**
- **High and low-level modules are grouped based on the control and data processing they provide for a specific program feature**
- Integration within the group progresses in alternating steps between the high and low level modules of the group
- **When integration for a certain functional group is complete, integration and testing moves onto the next group**
- Reaps the advantages of both types of integration while **minimizing the need for drivers and stubs**
- Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario



Validation testing

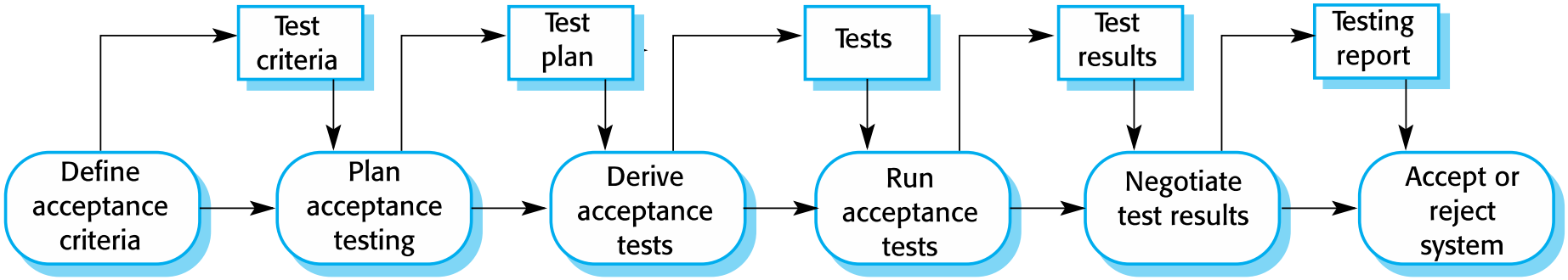
- **Validation testing follows integration testing**
- **Focuses on user-visible actions and user-recognizable output from the system**
- **Demonstrates conformity with requirements**
- **Designed to ensure that**
  - All functional requirements are satisfied
  - All behavioral characteristics are achieved
  - All performance requirements are attained
  - Documentation is correct
  - Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)
- **After each validation test**
  - The function or performance characteristic conforms to specification and is accepted
  - A deviation from specification is uncovered and a deficiency list is created

Types of user testing

- **Alpha testing**
  - Conducted at the developer's site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- **Beta testing**
  - Conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals

Acceptance testing

- **Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.**



Stages in the acceptance testing process

- Define acceptance criteria
- Plan acceptance testing
- Derive acceptance tests
- Run acceptance tests
- Negotiate test results
- Reject/accept system

Agile methods and acceptance testing

- In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- There is no separate acceptance testing process.
- Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.

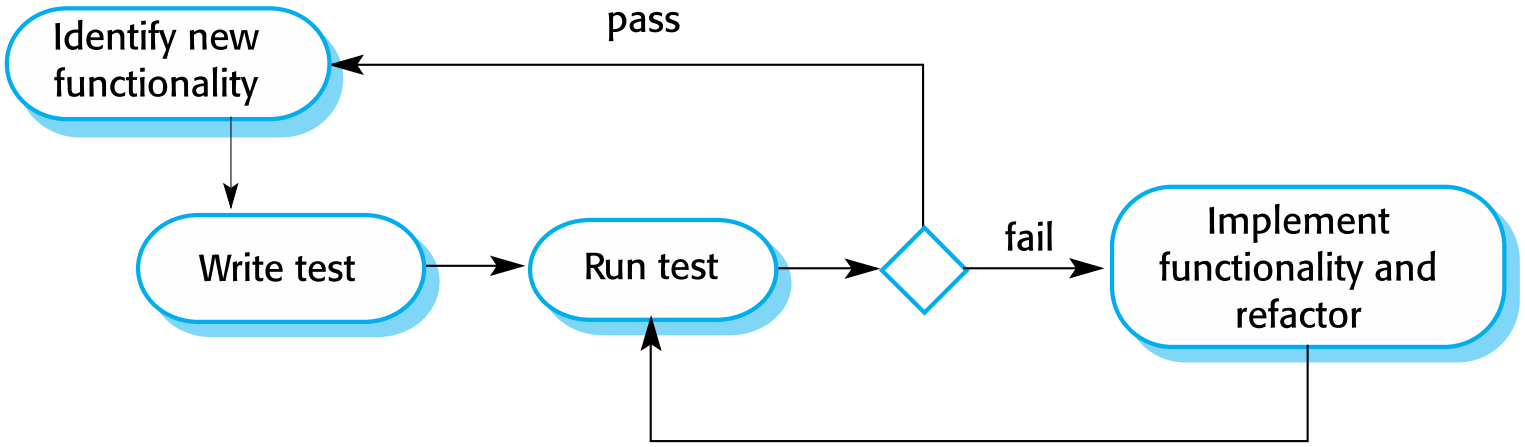
System testing

- **Recovery testing**
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- **Security testing**
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- **Stress testing**
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- **Performance testing**
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation
  - Can uncover situations that lead to degradation and possible system failure

Test-driven development

Test-driven development

- **Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.**
- **Tests are written before code** and 'passing' the tests is the critical driver of development.
- **You develop code incrementally, along with a test for that increment**. You don't move on to the next increment until the code that you have developed passes its test.
- **TDD was introduced as part of agile methods such as Extreme Programming**. **However, TDD can also be used in plan-driven development processes.**



TDD process activities

- **Start by identifying the increment of functionality** that is required. This should normally be small and implementable in **a few lines of code**.
- **Write a test for this functionality and implement this as an automated test**.
- **Run the test, along with all other tests that have been implemented**. Initially, you have not implemented the functionality so the new test will fail.
- **Implement the functionality and re-run the test**.
- **Once all tests run successfully**, you move on to implementing the next chunk of functionality.

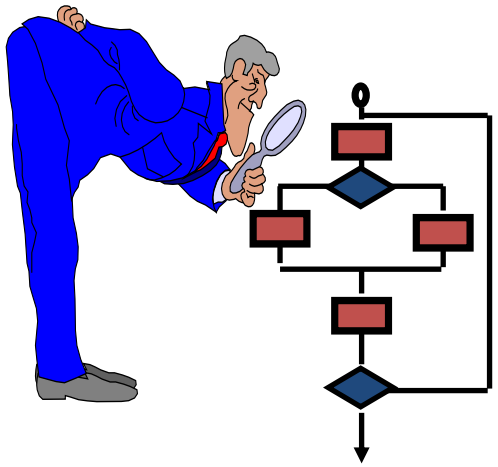
Benefits of TDD

- **Code coverage**
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- **Regression testing**
  - A regression test suite is developed incrementally as a program is developed.
- **Simplified debugging**
  - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- **System documentation**
  - The tests themselves are a form of documentation that describe what the code should be doing.

Software testing techniques

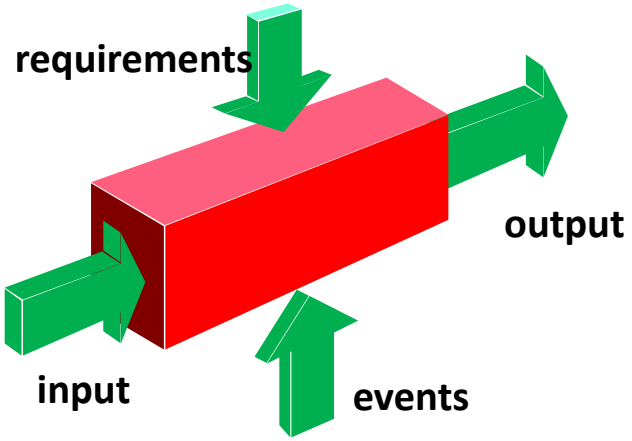
Software testing techniques

White-box techniques



- Intended to test internal paths and working of the software
- Examine internal program structure and derive tests from an examination of the program's logic.
- Used to develop test cases for unit and integration testing
- Other names: Glass-box, Logic-driven, Structural.

Black-box techniques



- Using the specifications of what the software should do
- Tests are derived from the I/O specification
- Used in most functional tests
- Other names: data-driven, input/output-driven.

White-box techniques

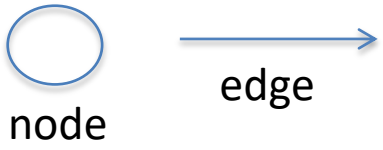
- Uses the control structure of the program/design to derive test cases
- We can derive test cases that:
  - Guarantee that all independent paths within a module have been visited at least once.
  - Exercise all logical decisions on their TRUE or FALSE sides
  - Execute all loops at their boundaries

Some White-box techniques

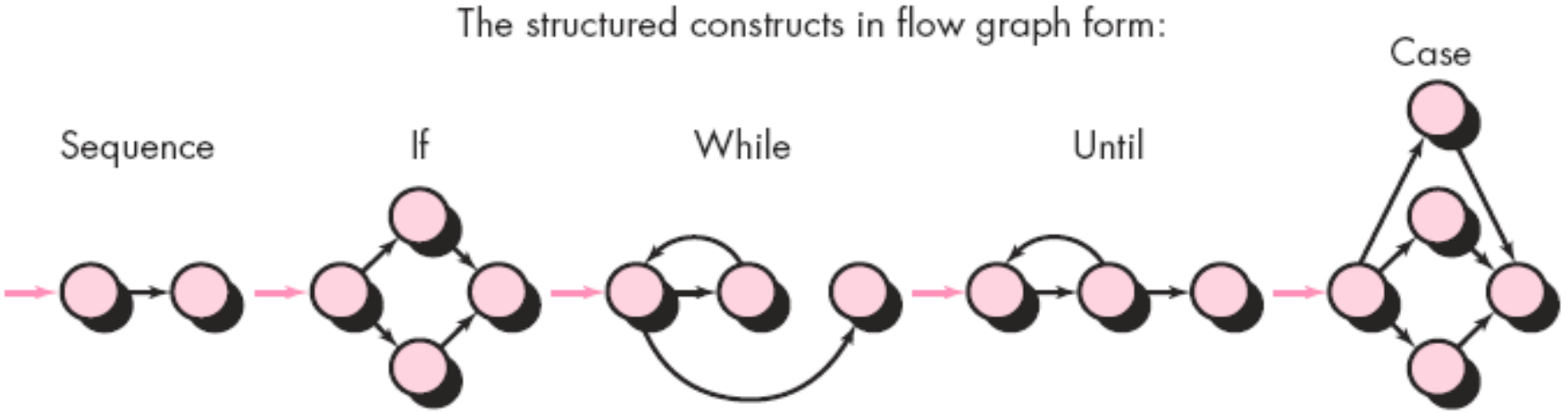
- Statement Coverage
  - Requires each statement of the program to be executed at least once.
  - Weakest of the white-box tests.
  - Considered as a minimum level of testing.
- Branch/Decision Coverage
  - Requires each branch to be traversed at least once.
  - More comprehensive than statement coverage.
- Basis Path
  - Execute all control flow paths through the code.

“Basis path” technique

- Proposed by Tom McCabe
- Execute all control flow paths through the code.
- Based on a flow graph.
  - Flow graph shows the logical control flow using following notation

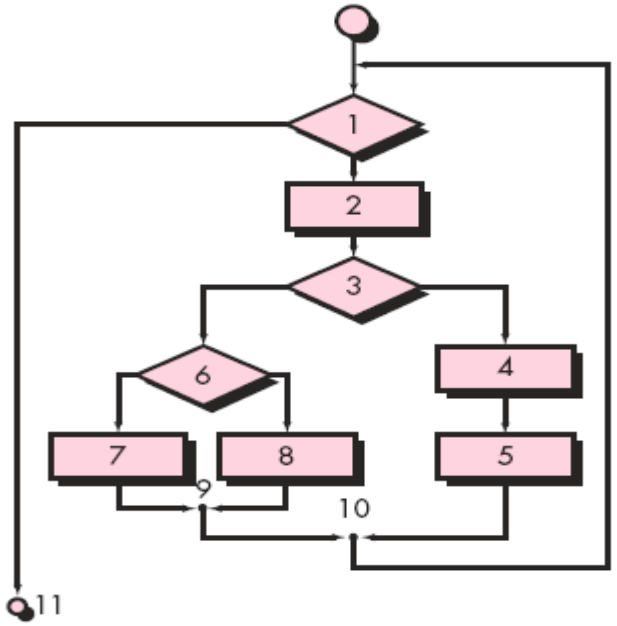


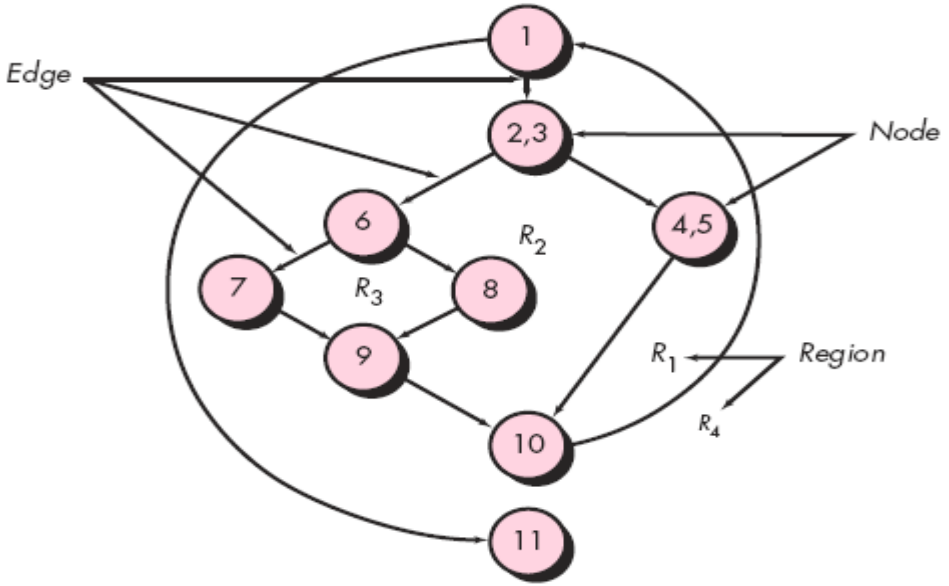
- An independent flow path is one that introduces at least 1 new set of statements or conditions
- Must move along at least 1 new edge on flow graph



“Basis path” technique

✧ From Flowchart to Flow graph





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

Set of independent paths for this example

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

“Basis path” technique

Cyclomatic complexity

- Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.
- When used in the context of the basis path testing method:
  - cyclomatic complexity defines the number of independent paths in the basis set of a program and
  - provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

- Cyclomatic complexity – V(G) is computed in one of three ways:
  1. V(G) = The number of regions of the flow graph
  2. V(G) = E - N + 2

     E is the number of flow graph edges
     N is the number of flow graph nodes
  3. V(G)=D + 1

     D is the number of decision nodes in the flow graph G.

For this example, V(G)=4

“Basis path” technique

Deriving test cases

- The following steps can be applied to derive the basis set:
  1. Using the design or code as a foundation, draw a corresponding flow graph.
  2. Determine the cyclomatic complexity of the resultant flow graph.
  3. Determine a basis set of linearly independent paths.
  4. Prepare test cases that will force execution of each path in the basis set.

- Preparation of test cases:
  - Data should be chosen so that conditions at the decision 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.

Black-box techniques

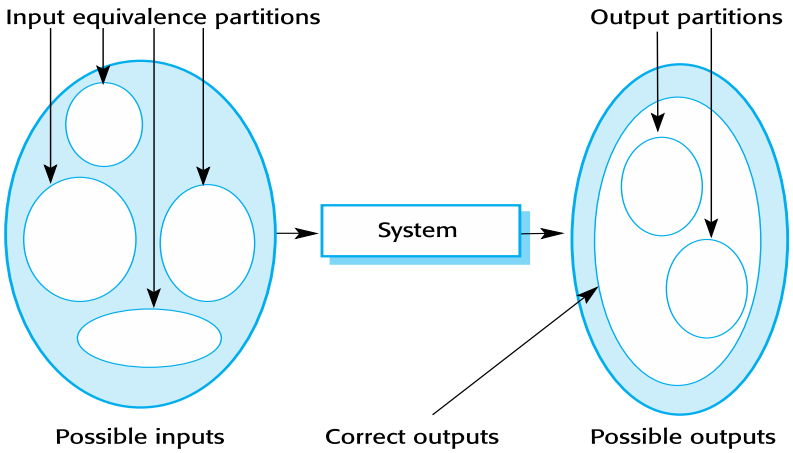
- **Black-box testing (also called behavioral testing), focuses on the functional software requirements without regard to the internal structure**
- **Black-box testing techniques enable us to derive sets of input conditions that will fully exercise all functional requirements.**
- **Black-box testing is not an alternative to white-box techniques – black-box is a complementary approach that is likely to uncover a different class of errors.**
- **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.**
- **Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.**
- **Some black-box techniques**
  - **Equivalence Partitioning**

    A black-box testing method that divides the input domain of a program into classes of data which test cases can be derived.
  - **Boundary Value Analysis,**

    A test case design technique that complements equivalence partitioning, by selecting test cases at the “edges” of the class

Equivalence Partitioning

- B-B testing method that divides the input domain into classes of data which test cases can be derived.
- **An equivalence class is a partition of domain data where the program behaves in an equivalent way for each class member.**
- **Test cases should be chosen from each partition**
- **Equivalence classes may be defined according to the following guidelines:**
  1. **If input condition specifies a range, one valid and two invalid equivalence classes** are defined.
  2. **If input condition requires a specific value, one valid and two invalid equivalence classes** are defined.
  3. **If input condition specifies a member of a set, one valid and one invalid equivalence class** are defined.
  4. **If input condition is Boolean, one valid and one invalid class** are defined.



- **Invalid data**
  - data outside bounds of the program
  - physically impossible data
  - proper value supplied in wrong place

Boundary Value Analysis

- **A greater number of errors occurs at the boundaries of the input domain rather than in the "center."**
- Boundary value analysis leads to a selection of test cases that exercise bounding values.
- Boundary value analysis is a test-case design technique that complements equivalence partitioning.
- **Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.**

- Guidelines for BVA are similar to those provided for equivalence partitioning:
  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. **Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.**