

# UNIT 5

## Software Testing Strategies

- A strategic approach to testing
- Test strategies for conventional software
- Validation testing
- System testing

# Introduction

- A strategy for software testing integrates the design of software test cases into a well-planned series of steps that result in successful development of the software
- The strategy provides a road map that describes the steps to be taken, when, and how much effort, time, and resources will be required
- The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation
- The strategy provides guidance for the practitioner and a set of milestones for the manager
- Because of time pressures, progress must be measurable and problems must surface as early as possible

# A Strategic Approach to Testing

# General Characteristics of Strategic Testing

- To perform effective testing, a software team should conduct effective formal technical reviews
- Testing begins at the component level and work outward toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects) by an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

# Verification and Validation

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- Verification (Are the algorithms coded correctly?)
  - The set of activities that ensure that software correctly implements a specific function or algorithm
- Validation (Does it meet user requirements?)
  - The set of activities that ensure that the software that has been built is traceable to customer requirements

## Verification

### • Definition

Verification refers to the set of activities that ensure software correctly implements the specific function.

### • Focus

It includes checking documents, designs, codes, and programs.



## Validation

### • Definition

Validation refers to the set of activities that ensure that the software that has been built is traceable to customer requirements.

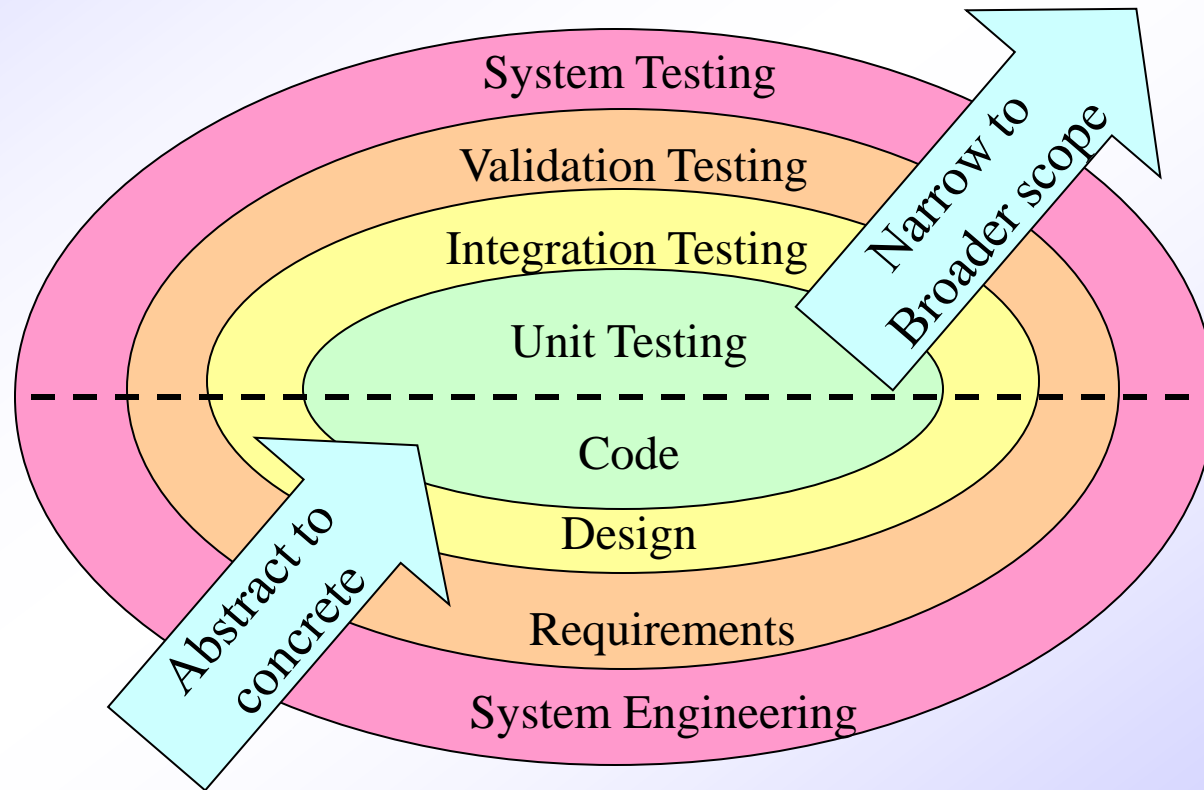
### • Focus

It includes testing and validating the actual product.

# Organizing for Software Testing

- 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

# A Strategy for Testing Conventional Software





# Levels of Testing for Conventional Software

- 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

# Testing Strategy applied to Conventional Software

- Unit testing
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
- Integration testing
  - Focuses on inputs and outputs, and how well the components fit together and work together
- Validation testing
  - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- System testing
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

# Testing Strategy applied to Object-Oriented Software

- Must broaden testing to include detections of errors in analysis and design models
- Unit testing loses some of its meaning and integration testing changes significantly
- Use the same philosophy but different approach as in conventional software testing
- Test "in the small" and then work out to testing "in the large"
  - Testing in the small involves class attributes and operations; the main focus is on communication and collaboration within the class
  - Testing in the large involves a series of regression tests to uncover errors due to communication and collaboration among classes
- Finally, the system as a whole is tested to detect errors in fulfilling requirements

# 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

# Ensuring a Successful Software Test Strategy

- Specify product requirements in a quantifiable manner long before testing commences
- State testing objectives explicitly in measurable terms
- Understand the user of the software (through use cases) and develop a profile for each user category
- Develop a testing plan that emphasizes rapid cycle testing to get quick feedback to control quality levels and adjust the test strategy
- Build robust software that is designed to test itself and can diagnose certain kinds of errors
- Use effective formal technical reviews as a filter prior to testing to reduce the amount of testing required
- Conduct formal technical reviews to assess the test strategy and test cases themselves
- Develop a continuous improvement approach for the testing process through the gathering of metrics

# Test Strategies for Conventional Software

# Unit Testing

- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- 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



# Common Computational Errors in Execution Paths

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (int vs. float)

# Other Errors to Uncover

- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely (using `==` with float types)
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

# Problems to uncover in Error Handling

- Error description is unintelligible or ambiguous
- Error noted does not correspond to error encountered
- Error condition causes operating system intervention prior to error handling
- Exception condition processing is incorrect
- Error description does not provide enough information to assist in the location of the cause of the error

# Drivers and Stubs for Unit Testing

- 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 subordinate to (called by) the component to be tested
  - It 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

# 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
  - Incremental Integration Testing

# Non-incremental Integration Testing

- Commonly called 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 Testing

- Three kinds
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration
- 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

- 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/testing process



# Bottom-up Integration

- 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

# Sandwich Integration

- Consists of a combination of both top-down and bottom-up integration
- 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

# 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
  - Ensures that changes have not propagated unintended side effects
  - 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

# Smoke Testing

- Taken from the world of hardware
  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

# Benefits of Smoke Testing

- Integration risk is minimized
  - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- The quality of the end-product is improved
  - Smoke testing is likely to uncover both functional errors and architectural and component-level design errors
- Error diagnosis and correction are simplified
  - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
  - As integration testing progresses, more software has been integrated and more has been demonstrated to work
  - Managers get a good indication that progress is being made

# Validation Testing

# Background

- Validation testing follows integration testing
- The distinction between conventional and object-oriented software disappears
- 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
- A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

# Alpha and Beta 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
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base



# System Testing

# Different Types

- 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

# Software Testing Techniques

- Testing fundamentals
- White-box testing
- Black-box testing

# 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

(more on next slide)

# Characteristics of Testable Software (continued)

- Simple
  - The program should exhibit functional, structural, and code simplicity
- Stable
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- Understandable
  - The architectural design is well understood; documentation is available and organized

# Test Characteristics

- A good test has a high probability of finding an error
  - The tester must understand the software and how it might fail
- A good test is not redundant
  - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be “best of breed”
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
  - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

# Two Unit Testing Techniques

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

# White-box Testing



# White-box Testing

- Uses the control structure part of component-level design to derive the test cases
- These test cases
  - Guarantee that all independent paths within a module have been exercised at least once
  - Exercise all logical decisions on their true and false sides
  - Execute all loops at their boundaries and within their operational bounds
  - Exercise internal data structures to ensure their validity

“Bugs lurk in corners and congregate at boundaries”

# Basis Path Testing

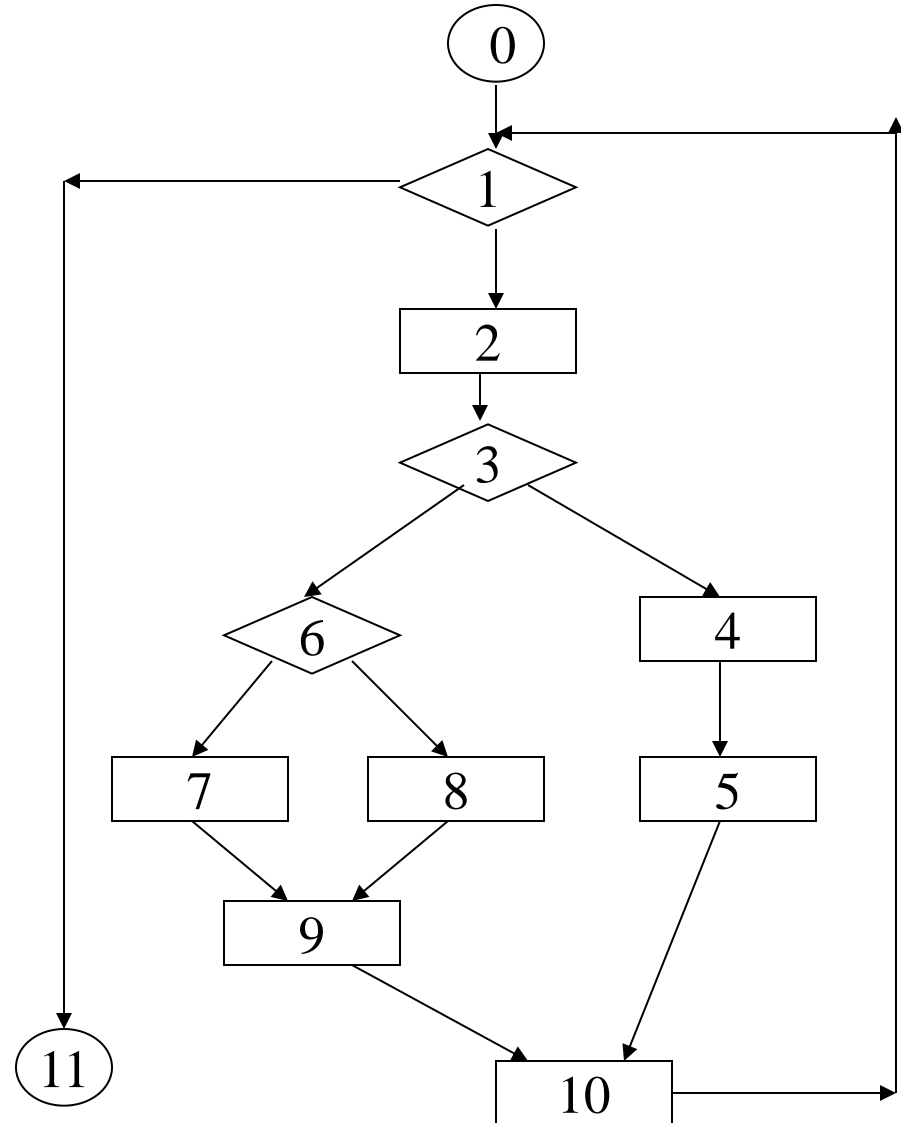
- White-box testing technique proposed by Tom McCabe
- Enables the test case designer to derive a logical complexity measure of a procedural design
- Uses this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise the basis set are guaranteed to execute every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

# Flow Graph Notation

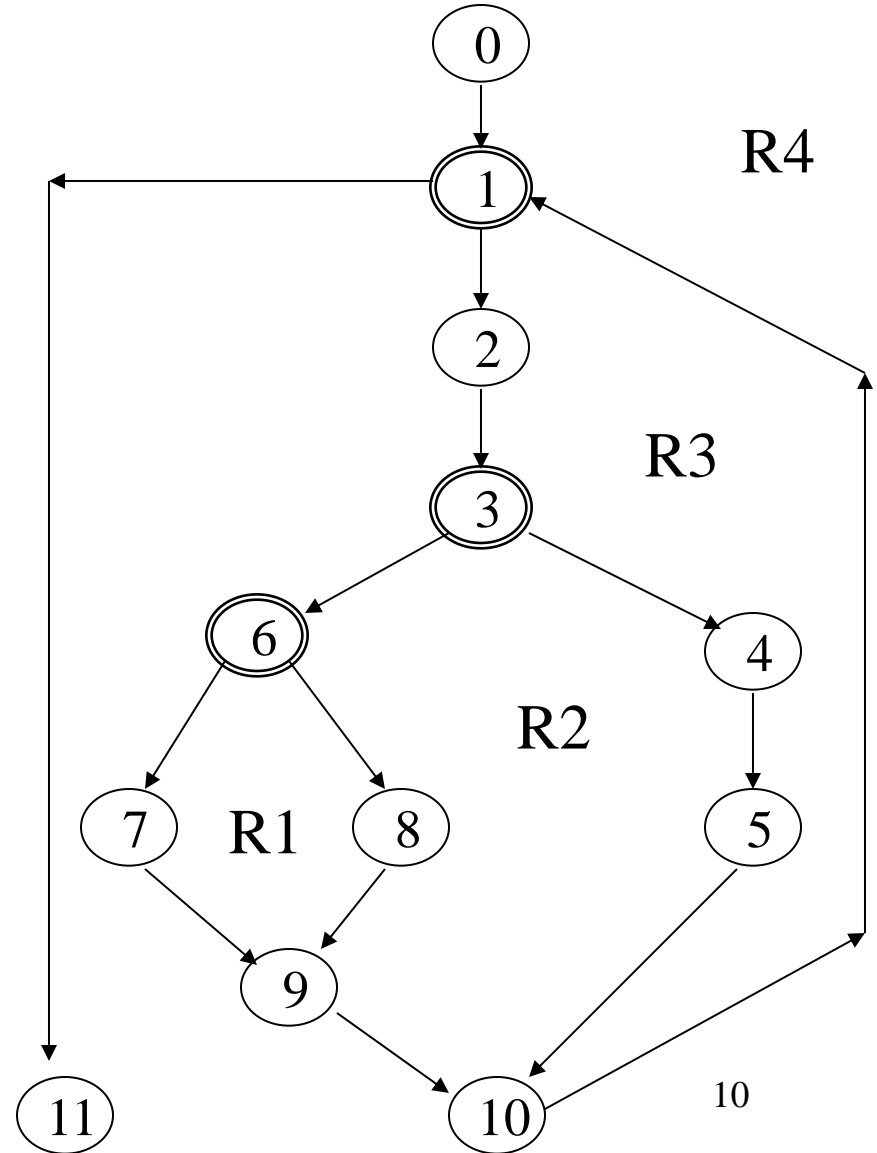
- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and,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

# Flow Graph Example

**FLOW CHART**



**FLOW GRAPH**



# Independent Program Paths

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

# Cyclomatic Complexity

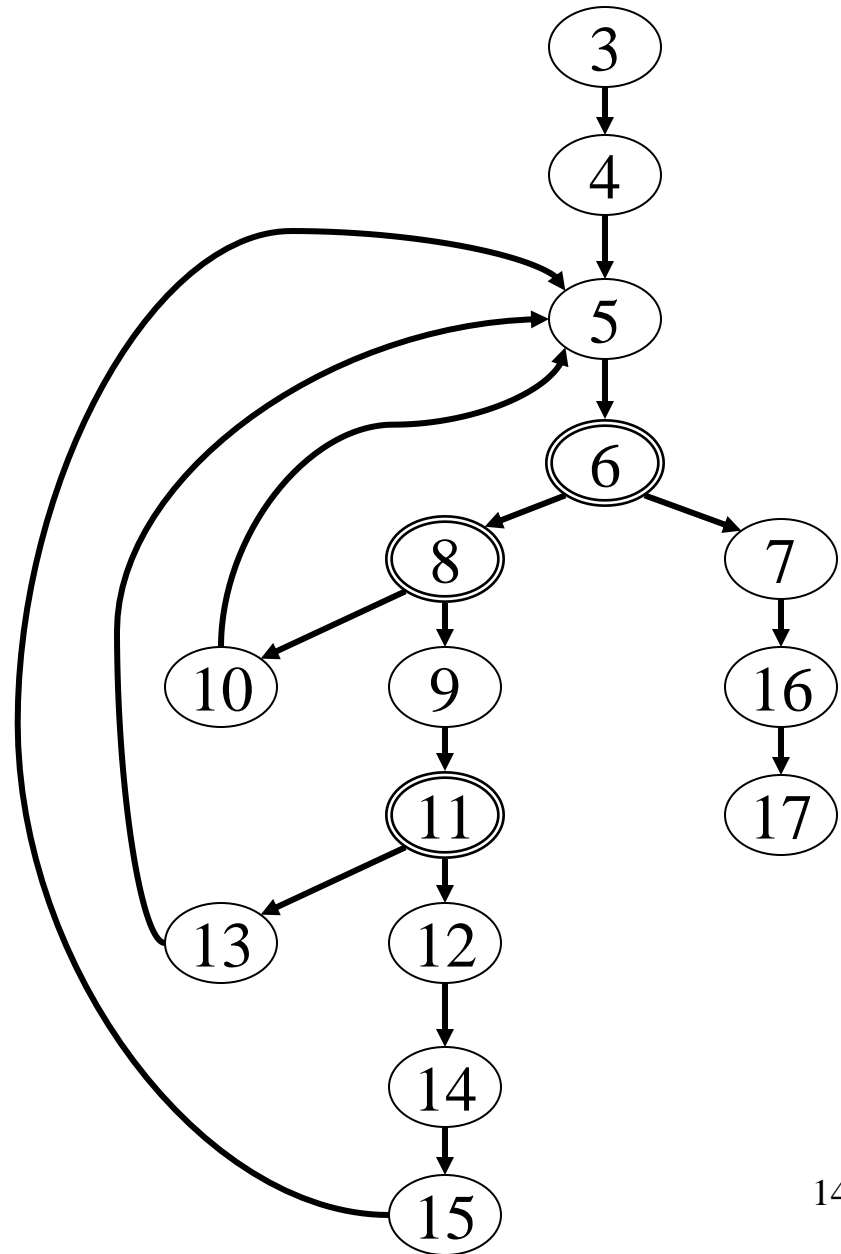
- Provides a quantitative measure of the logical complexity of a program
- Defines the number of independent paths in the basis set
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- Can be computed three ways
  - The number of regions
  - $V(G) = E - N + 2$ , where E is the number of edges and N is the number of nodes in graph G
  - $V(G) = P + 1$ , where P is the number of predicate nodes in the flow graph G
- Results in the following equations for the example flow graph
  - Number of regions = 4
  - $V(G) = 14 \text{ edges} - 12 \text{ nodes} + 2 = 4$
  - $V(G) = 3 \text{ predicate nodes} + 1 = 4$

# Deriving the Basis Set and Test Cases

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

# A Second Flow Graph Example

```
1  int functionY(void)
2  {
3      int x = 0;
4      int y = 19;
5  A: x++;
6      if (x > 999)
7          goto D;
8      if (x % 11 == 0)
9          goto B;
10     else goto A;
11 B: if (x % y == 0)
12     goto C;
13     else goto A;
14 C: printf("%d\n", x);
15     goto A;
16 D: printf("End of list\n");
17     return 0;
18 }
```





# A Sample Function to Diagram and Analyze

```
1  int functionZ(int y)
2  {
3  int x = 0;

4  while (x <= (y * y))
5      {
6          if ((x % 11 == 0) &&
7              (x % y == 0))
8              {
9                  printf("%d", x);
10                 x++;
11             } // End if
12         else if ((x % 7 == 0) ||
13                 (x % y == 1))
14             {
15                 printf("%d", y);
16                 x = x + 2;
17             } // End else
18         printf("\n");
19     } // End while

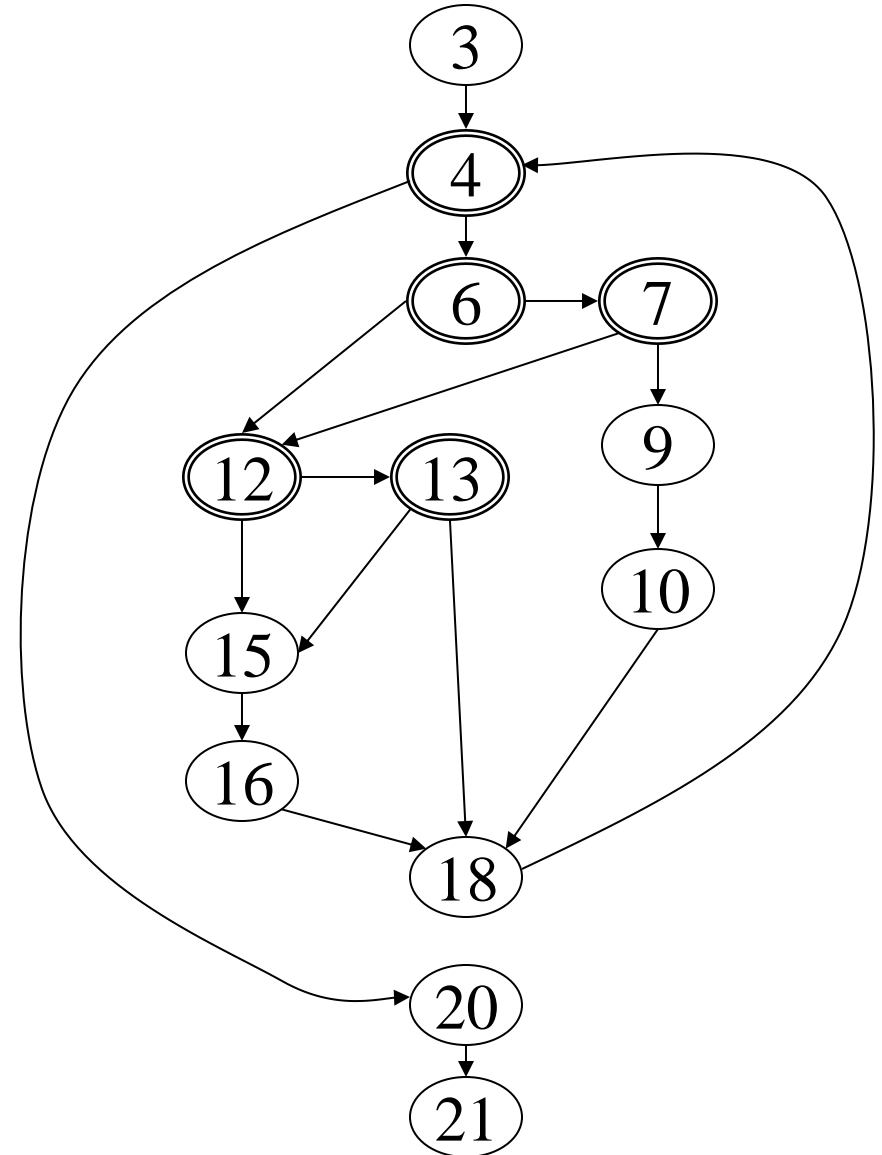
20 printf("End of list\n");
21 return 0;
22 } // End functionZ
```

# A Sample Function to Diagram and Analyze

```
1  int functionZ(int y)
2  {
3  int x = 0;

4  while (x <= (y * y))
5      {
6      if ((x % 11 == 0) &&
7          (x % y == 0))
8          {
9          printf("%d", x);
10         x++;
11         } // End if
12     else if ((x % 7 == 0) ||
13              (x % y == 1))
14         {
15         printf("%d", y);
16         x = x + 2;
17         } // End else
18     printf("\n");
19 } // End while

20 printf("End of list\n");
21 return 0;
22 } // End functionZ
```



# Loop Testing - General

- A white-box testing technique that focuses exclusively on the validity of loop constructs
- Four different classes of loops exist
  - Simple loops
  - Nested loops
  - Concatenated loops
  - Unstructured loops
- Testing occurs by varying the loop boundary values
  - Examples:

```
for (i = 0; i < MAX_INDEX; i++)
```

```
while (currentTemp >= MINIMUM_TEMPERATURE)
```

# Testing of Simple Loops

- 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

‘ $n$ ’ is the maximum number of allowable passes through the loop

# Testing of Nested Loops

- 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 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

# Testing of Concatenated Loops

- For independent loops, use the same approach as for simple loops
- Otherwise, use the approach applied for nested loops

# Testing of Unstructured Loops

- Redesign the code to reflect the use of structured programming practices
- Depending on the resultant design, apply testing for simple loops, nested loops, or concatenated loops

# Black-box Testing



# Black-box Testing

- Complements white-box testing by uncovering different classes of errors
- Focuses on the functional requirements and the information domain of the software
- Used during the later stages of testing after white box testing has been performed
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program
- The test cases satisfy the following:
  - Reduce, by a count greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  - Tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific task at hand

# Black-box Testing Categories

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external data base access
- Behavior or performance errors
- Initialization and termination errors

# Questions answered by Black-box Testing

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundary values of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

# Equivalence Partitioning

- A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived
- An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed
- Test case design is based on an evaluation of equivalence classes for an input condition
- An equivalence class represents a set of valid or invalid states for input conditions
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercised at once

# Guidelines for Defining Equivalence Classes

- If an input condition specifies a range, one valid and two invalid equivalence classes are defined
  - Input range: 1 – 10                      Eq classes: {1..10}, {x < 1}, {x > 10}
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  - Input value: 250                      Eq classes: {250}, {x < 250}, {x > 250}
- If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4}              Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an input condition is a Boolean value, one valid and one invalid class are defined
  - Input: {true condition}              Eq classes: {true condition}, {false condition}

# Boundary Value Analysis

- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
  - It selects test cases at the edges of a class
  - It derives test cases from both the input domain and output domain

# Guidelines for Boundary Value Analysis

- 1. If an input condition specifies a range bounded by values  $a$  and  $b$ , test cases should be designed with values  $a$  and  $b$  as well as values just above and just below  $a$  and  $b$
- 2. If an input condition specifies a number of values, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested
- Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above
- If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries