Unit 5

Software Testing Strategies

- 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

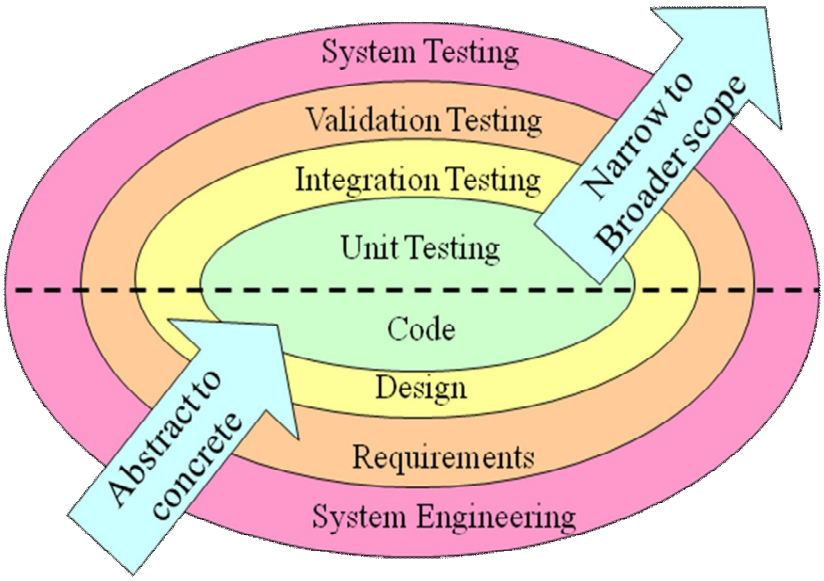
Organizing for Software Testing

- 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

Test Strategies for Object-Oriented Software

- With object-oriented software, you can no longer test a single operation in isolation (conventional thinking)
- Traditional top-down or bottom-up integration testing has little meaning

- Class testing for object-oriented software is the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class
- Drivers can be used
  - To test operations at the lowest level and for testing whole groups of classes
  - To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface
- Stubs can be used
  - In situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented
- Two different object-oriented testing strategies
  - Thread-based testing
    - Integrates the set of classes required to respond to one input or event for the system
    - Each thread is integrated and tested individually
    - Regression testing is applied to ensure that no side effects occur
  - Use-based testing
    - First tests the independent classes that use very few, if any, server classes
    - Then the next layer of classes, called dependent classes, are integrated
    - This sequence of testing layer of dependent classes continues until the entire system is constructed

Validation Testing

- 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

The Art of Debugging

Debugging Process

- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction

Why is Debugging so Difficult?

- The symptom and the cause may be geographically remote
- The symptom may disappear (temporarily) when another error is corrected
- The symptom may actually be caused by nonerrors (e.g., round-off accuracies)
- The symptom may be caused by human error that is not easily traced
- The symptom may be a result of timing problems, rather than processing problems
- It may be difficult to accurately reproduce input conditions, such as asynchronous real-time information
- The symptom may be intermittent such as in embedded systems involving both hardware and software
- The symptom may be due to causes that are distributed across a number of tasks running on different processes

Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies
  - **Brute force**
  - **Backtracking**
  - **Cause elimination**

Strategy #1: Brute Force

- Most commonly used and least efficient method

- Used when all else fails
- Involves the use of memory dumps, run-time traces, and output statements
- Leads many times to wasted effort and time

Strategy #2: Backtracking

- Can be used successfully in small programs
- The method starts at the location where a symptom has been uncovered
- The source code is then traced backward (manually) until the location of the cause is found
- In large programs, the number of potential backward paths may become unmanageably large

Strategy #3: Cause Elimination

- Involves the use of induction or deduction and introduces the concept of binary partitioning
  - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
  - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
- Data related to the error occurrence are organized to isolate potential causes
- A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis
- Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

Three Questions to ask Before Correcting the Error

- Is the cause of the bug reproduced in another part of the program?
  - Similar errors may be occurring in other parts of the program
- What next bug might be introduced by the fix that I'm about to make?
  - The source code (and even the design) should be studied to assess the coupling of logic and data structures related to the fix
- What could we have done to prevent this bug in the first place?
  - This is the first step toward software quality assurance
  - By correcting the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs

Software Testing Techniques

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

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

- 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

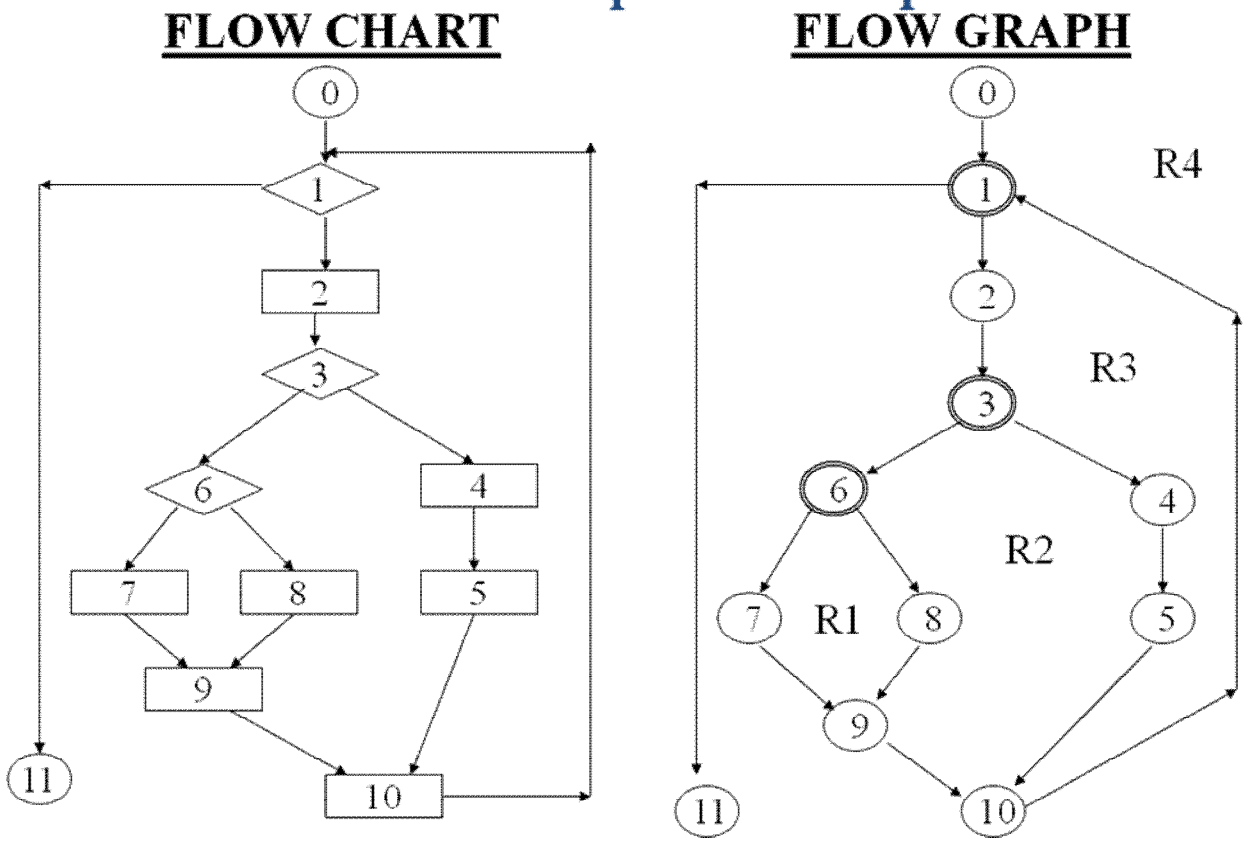
Basis Path Testing

- 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



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 edges – 12 nodes + 2 = 4
  - V(G) = 3 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

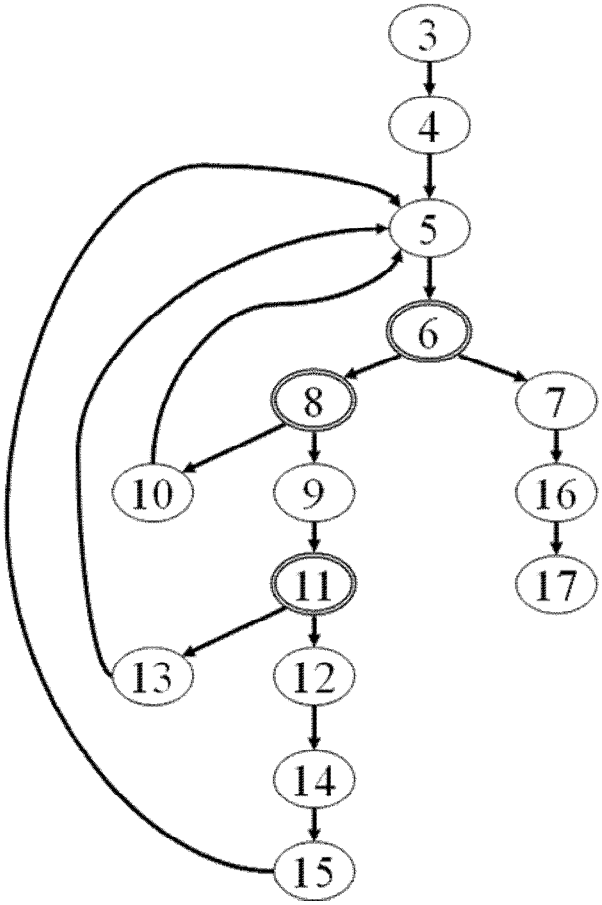
```
 1  int functionY(void)
 2  {
 3     int x = 0;
 4     int y = 19;

 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  }
```

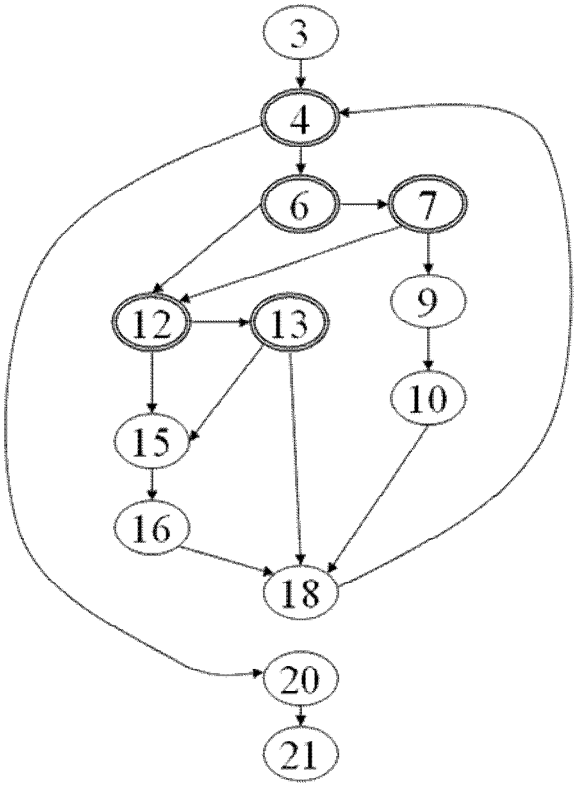


A Sample Function to Diagram and Analyze

```
 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
```



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

- 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 exercise 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 define
  - 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

Object-Oriented Testing Methods

- It is necessary to test an object-oriented system at a variety of different levels
- The goal is to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers
  - Testing begins "in the small" on methods within a class and on collaboration between classes
  - As class integration occurs, use-based testing and fault-based testing are applied
  - Finally, use cases are used to uncover errors during the software validation phase
- Conventional test case design is driven by an input-process-output view of software
- Object-oriented testing focuses on designing appropriate sequences of methods to exercise the states of a class

Testing Implications for Object-Oriented Software

- Because attributes and methods are encapsulated in a class, testing methods from outside of a class is generally unproductive
- Testing requires reporting on the state of an object, yet encapsulation can make this information somewhat difficult to obtain
- Built-in methods should be provided to report the values of class attributes in order to get a snapshot of the state of an object
- Inheritance requires retesting of each new context of usage for a class
  - If a subclass is used in an entirely different context than the super class, the super class test cases will have little applicability and a new set of tests must be designed

Applicability of Conventional Testing Methods

- White-box testing can be applied to the operations defined in a class
  - Basis path testing and loop testing can help ensure that every statement in an method has been tested
- Black-box testing methods are also appropriate
  - Use cases can provide useful input in the design of black-box tests

Fault-based Testing

- The objective in fault-based testing is to design tests that have a high likelihood of uncovering plausible faults
- Fault-based testing begins with the analysis model
  - The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects)
  - To determine whether these faults exist, test cases are designed to exercise the design or code
- If the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find a significant number of errors
- Integration testing looks for plausible faults in method calls or message connections (i.e., client/server exchange)
- Three types of faults are encountered in this context
  - Unexpected result
  - Wrong method or message used
  - Incorrect invocation
- The behavior of a method must be examined to determine the occurrence of plausible faults as methods are invoked
- Testing should exercise the attributes of an object to determine whether proper values occur for distinct types of object behavior
- The focus of integration testing is to determine whether errors exist in the calling code, not the called code

Fault-based Testing vs. Scenario-based Testing

- Fault-based testing misses two main types of errors
  - Incorrect specification: subsystem doesn't do what the user wants
  - Interactions among subsystems: behavior of one subsystem creates circumstances that cause another subsystem to fail
- A solution to this problem is scenario-based testing
  - It concentrates on what the user does, not what the product does
  - This means capturing the tasks (via use cases) that the user has to perform, then applying them as tests

- Scenario-based testing tends to exercise multiple subsystems in a single test

Random Order Testing (at the Class Level)

- Certain methods in a class may constitute a minimum behavioral life history of an object (e.g., open, seek, read, close); consequently, they may have implicit order dependencies or expectations designed into them
- Using the methods for a class, a variety of method sequences are generated randomly and then executed
- The goal is to detect these order dependencies or expectations and make appropriate adjustments to the design of the methods

Partition Testing (at the Class Level)

- Similar to equivalence partitioning for conventional software
- Methods are grouped based on one of three partitioning approaches
- State-based partitioning categorizes class methods based on their ability to change the state of the class
  - Tests are designed in a way that exercise methods that change state and those that do not change state
- Attribute-based partitioning categorizes class methods based on the attributes that they use
  - Methods are partitioned into those that read an attribute, modify an attribute, or do not reference the attribute at all
- Category-based partitioning categorizes class methods based on the generic function that each performs
  - Example categories are initialization methods, computational methods, and termination methods

Multiple Class Testing

- Class collaboration testing can be accomplished by applying random testing, partition testing, scenario-based testing and behavioral testing
- The following sequence of steps can be used to generate multiple class random test cases
  1) For each client class, use the list of class methods to generate a series of random test sequences; use these methods to send messages to server classes
  2) For each message that is generated, determine the collaborator class and the corresponding method in the server object
  3) For each method in the server object (invoked by messages from the client object), determine the messages that it transmits
  4) For each of these messages, determine the next level of methods that are invoked and incorporate these into the test sequence

Tests Derived from Behavior Models

- The state diagram for a class can be used to derive a sequence of tests that will exercise the dynamic behavior of the class and the classes that collaborate with it

- The test cases should be designed to achieve coverage of all states
  - Method sequences should cause the object to transition through all allowable states
- More test cases should be derived to ensure that all behaviors for the class have been exercised based on the behavior life history of the object
- The state diagram can be traversed in a "breadth-first" approach by exercising only a single transition at a time
  - When a new transition is to be tested, only previously tested transitions are used

- - - - - - - - - - - - - - -

Software Product Metrics

Examples of Metrics from Everyday Life

- Working and living
  - Cost of utilities for the month
  - Cost of groceries for the month
  - Amount of monthly rent per month
  - Time spent at work each Saturday for the past month
  - Time spent mowing the lawn for the past two times
- College experience
  - Grades received in class last semester
  - Number of classes taken each semester
  - Amount of time spent in class this week
  - Amount of time spent on studying and homework this week
  - Number of hours of sleep last night
- Travel
  - Time to drive from home to the airport
  - Amount of miles traveled today
  - Cost of meals and lodging for yesterday

Why have Software Product Metrics?

- Help software engineers to better understand the attributes of models and assess the quality of the software
- Help software engineers to gain insight into the design and construction of the software
- Focus on specific attributes of software engineering work products resulting from analysis, design, coding, and testing

- Provide a systematic way to assess quality based on a set of clearly defined rules
- Provide an "on-the-spot" rather than "after-the-fact" insight into the software development

Software Quality

- Definition:

  Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software

- Three important points in this definition
  - Explicit software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality
  - Specific standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will most surely result
  - There is a set of implicit requirements that often goes unmentioned (e.g., ease of use). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect

Properties of Software Quality Factors

- Some factors can be directly measured (e.g. defects uncovered during testing)
- Other factors can be measured only indirectly (e.g., usability or maintainability)
- Software quality factors can focus on three important aspects
  - Product operation: Its operational characteristics
  - Product revision: Its ability to undergo change
  - Product transition: Its adaptability to new environments

ISO 9126 Software Quality Factors

- Functionality
  - The degree to which the software satisfies stated needs
- Reliability
  - The amount of time that the software is available for use
- Usability
  - The degree to which the software is easy to use
- Efficiency
  - The degree to which the software makes optimal use of system resources
- Maintainability

  - The ease with which repair and enhancement may be made to the software
- Portability
  - The ease with which the software can be transposed from one environment to another

A Framework for Product Metrics

Measures, Metrics, and Indicators

- These three terms are often used interchangeably, but they can have subtle differences
- **Measure**
  - Provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
- **Measurement**
  - The act of determining a measure
- **Metric**
  - (IEEE) A quantitative measure of the degree to which a system, component, or process possesses a given attribute
- **Indicator**
  - A metric or combination of metrics that provides insight into the software process, a software project, or the product itself

Purpose of Product Metrics

- Aid in the evaluation of analysis and design models
- Provide an indication of the complexity of procedural designs and source code
- Facilitate the design of more effective testing techniques
- Assess the stability of a fielded software product

Activities of a Measurement Process

- Formulation
  - The derivation (i.e., identification) of software measures and metrics appropriate for the representation of the software that is being considered
- Collection
  - The mechanism used to accumulate data required to derive the formulated metrics
- Analysis
  - The computation of metrics and the application of mathematical tools
- Interpretation

  - The evaluation of metrics in an effort to gain insight into the quality of the representation
- Feedback
  - Recommendations derived from the interpretation of product metrics and passed on to the software development team

Characterizing and Validating Metrics

- A metric should have desirable mathematical properties
  - It should have a meaningful range (e.g., zero to ten)
  - It should not be set on a rational scale if it is composed of components measured on an ordinal scale
- If a metric represents a software characteristic that increases when positive traits occur or decreases when undesirable traits are encountered, the value of the metric should increase or decrease in the same manner
- Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions
  - It should measure the factor of interest independently of other factors
  - It should scale up to large systems
  - It should work in a variety of programming languages and system domains

Collection and Analysis Guidelines

- Whenever possible, data collection and analysis should be automated
- Valid statistical techniques should be applied to establish relationships between internal product attributes and external quality characteristics
- Interpretative guidelines and recommendations should be established for each metric

Goal-oriented Software Measurement

- Goal/Question/Metric (GQM) paradigm
- GQM technique identifies meaningful metrics for any part of the software process
- GQM emphasizes the need to
  - Establish an explicit measurement goal that is specific to the process activity or product characteristic that is to be assessed
  - Define a set of questions that must be answered in order to achieve the goal
  - Identify well-formulated metrics that help to answer these questions
- GQM utilizes a goal definition template to define each measurement goal
- Example use of goal definition template

  Analyze the SafeHome software architecture for the purpose of evaluating architecture components. Do this with respect to the ability to make SafeHome more extensible from the viewpoint of the software engineers, who are performing the work in the context of product enhancement over the next three years.

- Example questions for this goal definition
  - Are architectural components characterized in a manner that compartmentalizes function and related data?
  - Is the complexity of each component within bounds that will facilitate modification and extension?

Attributes of Effective Software Metrics

- Simple and computable
  - It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time
- Empirically and intuitively persuasive
  - The metric should satisfy the engineer's intuitive notions about the product attribute under consideration
- Consistent and objective
  - The metric should always yield results that are unambiguous
- Consistent in the use of units and dimensions
  - The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units
- Programming language independent
  - Metrics should be based on the analysis model, the design model, or the structure of the program itself
- An effective mechanism for high-quality feedback
  - The metric should lead to a higher-quality end product

A Product Metrics Taxonomy

Metrics for the Analysis Model

- Functionality delivered
  - Provides an indirect measure of the functionality that is packaged within the software
- System size
  - Measures the overall size of the system defined in terms of information available as part of the analysis model
- Specification quality
  - Provides an indication of the specificity and completeness of a requirements specification

Eg : Function Points

- First proposed by Albrecht in 1979; hundreds of books and papers have been written on functions points since then

- Can be used effectively as a means for measuring the functionality delivered by a system
- Using historical data, function points can be used to
  - Estimate the cost or effort required to design, code, and test the software
  - Predict the number of errors that will be encountered during testing
  - Forecast the number of components and/or the number of projected source code lines in the implemented system
- Derived using an empirical relationship based on
  - Countable (direct) measures of the software's information domain
  - Assessments of the software's complexity

Information Domain Values

- Number of external inputs
  - Each external input originates from a user or is transmitted from another application
  - They provide distinct application-oriented data or control information
  - They are often used to update internal logical files
  - They are not inquiries (those are counted under another category)
- Number of external outputs
  - Each external output is derived within the application and provides information to the user
  - This refers to reports, screens, error messages, etc.
  - Individual data items within a report or screen are not counted separately
- Number of external inquiries
  - An external inquiry is defined as an online input that results in the generation of some immediate software response
  - The response is in the form of an on-line output
- Number of internal logical files
  - Each internal logical file is a logical grouping of data that resides within the application's boundary and is maintained via external inputs
- Number of external interface files
  - Each external interface file is a logical grouping of data that resides external to the application but provides data that may be of use to the application

Function Point Computation

1) Identify/collect the information domain values
2) Complete the table shown below to get the count total

- Associate a weighting factor (i.e., complexity value) with each count based on criteria established by the software development organization

3) Evaluate and sum up the adjustment factors (see the next two slides)
   - "F_i" refers to 14 value adjustment factors, with each ranging in value from 0 (not important) to 5 (absolutely essential)

4) Compute the number of function points (FP)
   FP = count total * [0.65 + 0.01 * sum(F_i)]

Information Domain Value	Count		Weighting Factor Simple	Average	Complex	
External Inputs	_____	x	3	4	6	= _____
External Outputs	_____	x	4	5	7	= _____
External Inquiries	_____	x	3	4	6	= _____
Internal Logical Files	_____	x	7	10	15	= _____
External Interface Files	_____	x	5	7	10	= _____
Count total						_____

Value Adjustment Factors

1) Does the system require reliable backup and recovery?
2) Are specialized data communications required to transfer information to or from the application?
3) Are there distributed processing functions?
4) Is performance critical?
5) Will the system run in an existing, heavily utilized operational environment?
6) Does the system require on-line data entry?
7) Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8) Are the internal logical files updated on-line?
9) Are the inputs, outputs, files, or inquiries complex?
10) Is the internal processing complex?
11) Is the code designed to be reusable?
12) Are conversion and installation included in the design?
13) Is the system designed for multiple installations in different organizations?
14) Is the application designed to facilitate change and for ease of use by the user?

Function Point Example

Information Domain Value	Count		Weighting Factor Simple	Average	Complex	
External Inputs	3	x	3	4	6	= 9
External Outputs	2	x	4	5	7	= 8
External Inquiries	2	x	3	4	6	= 6
Internal Logical Files	1	x	7	10	15	= 7
External Interface Files	4	x	5	7	10	= 20
Count total						**50**

- FP = count total * [0.65 + 0.01 * sum(F_i)]
- FP = 50 * [0.65 + (0.01 * 46)]
- FP = 55.5 (rounded up to 56)

Interpretation of the FP Number

- Assume that past project data for a software development group indicates that
  - One FP translates into 60 lines of object-oriented source code
  - 12 FPs are produced for each person-month of effort
  - An average of three errors per function point are found during analysis and design reviews
  - An average of four errors per function point are found during unit and integration testing
- This data can help project managers revise their earlier estimates
- This data can also help software engineers estimate the overall implementation size of their code and assess the completeness of their review and testing activities

Metrics for the Design Model

- Architectural metrics
  - Provide an indication of the quality of the architectural design
- Component-level metrics
  - Measure the complexity of software components and other characteristics that have a bearing on quality
- Interface design metrics
  - Focus primarily on usability
- Specialized object-oriented design metrics
  - Measure characteristics of classes and their communication and collaboration characteristics

Architectural Design Metrics

- These metrics place emphasis on the architectural structure and effectiveness of modules or components within the architecture
- They are “black box” in that they do not require any knowledge of the inner workings of a particular software component

Hierarchical Architecture Metrics

- Fan out: the number of modules immediately subordinate to the module *i*, that is, the number of modules directly invoked by module *i*
- Structural complexity

  - $S(i) = f^2_{out}(i)$, where $f_{out}(i)$ is the "fan out" of module *i*
- Data complexity
  - $D(i) = v(i)/[f_{out}(i) + 1]$, where $v(i)$ is the number of input and output variables that are passed to and from module *i*
- System complexity
  - $C(i) = S(i) + D(i)$
- As each of these complexity values increases, the overall architectural complexity of the system also increases
- This leads to greater likelihood that the integration and testing effort will also increase
- Shape complexity
  - size = n + a, where n is the number of nodes and a is the number of arcs
  - Allows different program software architectures to be compared in a straightforward manner
- Connectivity density (i.e., the arc-to-node ratio)
  - r = a/n
  - May provide a simple indication of the coupling in the software architecture

Metrics for Object-Oriented Design

- Size
  - Population: a static count of all classes and methods
  - Volume: a dynamic count of all instantiated objects at a given time
  - Length: the depth of an inheritance tree
- Coupling
  - The number of collaborations between classes or the number of methods called between objects
- Cohesion
  - The cohesion of a class is the degree to which its set of properties is part of the problem or design domain
- Primitiveness
  - The degree to which a method in a class is atomic (i.e., the method cannot be constructed out of a sequence of other methods provided by the class)

Specific Class-oriented Metrics

- Weighted methods per class
  - The normalized complexity of the methods in a class
  - Indicates the amount of effort to implement and test a class

- Depth of the inheritance tree
  - The maximum length from the derived class (the node) to the base class (the root)
  - Indicates the potential difficulties when attempting to predict the behavior of a class because of the number of inherited methods
- Number of children (i.e., subclasses)
  - As the number of children of a class grows
    - Reuse increases
    - The abstraction represented by the parent class can be diluted by inappropriate children
    - The amount of testing required will increase
- Coupling between object classes
  - Measures the number of collaborations a class has with any other classes
  - Higher coupling decreases the reusability of a class
  - Higher coupling complicates modifications and testing
  - Coupling should be kept as low as possible
- Response for a class
  - This is the set of methods that can potentially be executed in a class in response to a public method call from outside the class
  - As the response value increases, the effort required for testing also increases as does the overall design complexity of the class
- Lack of cohesion in methods
  - This measures the number of methods that access one or more of the same instance variables (i.e., attributes) of a class
  - If no methods access the same attribute, then the measure is zero
  - As the measure increases, methods become more coupled to one another via attributes, thereby increasing the complexity of the class design

Metrics for Source Code

- Complexity metrics
  - Measure the logical complexity of source code (can also be applied to component-level design)
- Length metrics
  - Provide an indication of the size of the software

"These metrics can be used to assess source code complexity, maintainability, and testability, among other characteristics"

Metrics for Testing

- Statement and branch coverage metrics
  - Lead to the design of test cases that provide program coverage
- Defect-related metrics
  - Focus on defects (i.e., bugs) found, rather than on the tests themselves
- Testing effectiveness metrics
  - Provide a real-time indication of the effectiveness of tests that have been conducted
- In-process metrics
  - Process related metrics that can be determined as testing is conducted

Metrics for Maintenance

- Software maturity index (SMI)
  - Provides an indication of the stability of a software product based on changes that occur for each release
- SMI = $[M_T - (F_a + F_c + F_d)]/M_T$
  where
  M_T = #modules in the current release
  F_a = #modules in the current release that have been added
  F_c = #modules in the current release that have been changed
  F_d = #modules from the preceding release that were deleted in the current release
- As the SMI (i.e., the fraction) approaches 1.0, the software product begins to stabilize
- The average time to produce a release of a software product can be correlated with the SMI