Tactics and Techniques of Software Testing

Software Testing Techniques

- Testing fundamentals
- White-box testing
- Black-box testing
- Object-oriented testing methods
- Manual vs Automated Testing
- The Art of Debugging

Characteristics of Testable Software

(as per James Bach)

- 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

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

Delivered Defects, Reliability, Customer Satisfaction

Delivered Defects per KLOC	Defects per Function Point	Mean Time to Failure (MTTF hours)	Customer Satisfaction
0.00	0.00	Infinite	Excellent
1.00	0.13	303	Very good
2.00	0.25	223	Good
3.00	0.38	157	Fair
4.00	0.50	105	Poor
5.00	0.63	66	Very Poor
7.00	0.88	17	Very Poor
9.00	1.13	1	Litigation
10.00	1.25	<1	Malpractice

Capers Jones considered C projects, assumes 125 LOC per FP

Capers Jones., Software Engineering Best Practices, TMH ©2010

Test Characteristics

(as per Kaner et al)

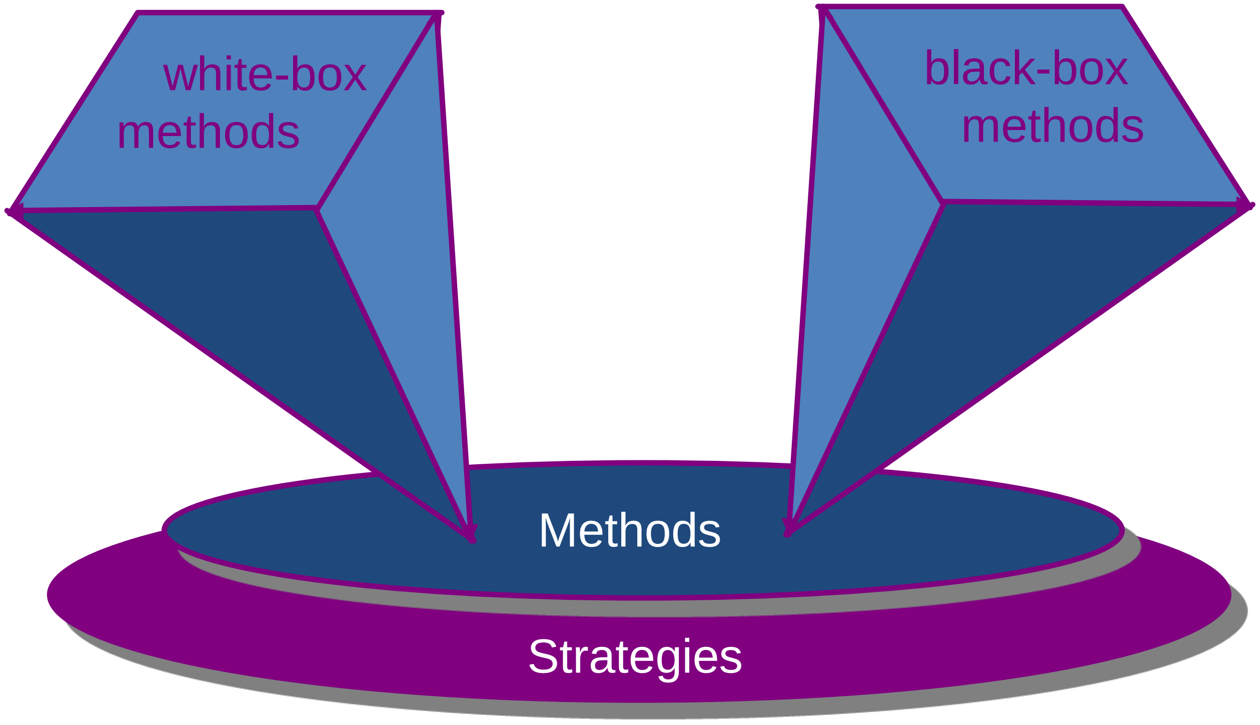
- A good test has a high probability of finding an error
  - The tester must understand the software and how it might fail
- A good test is not redundant
  - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be “best of breed”
  - Tests that have 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

Software Testing



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 <u>all independent paths</u> 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

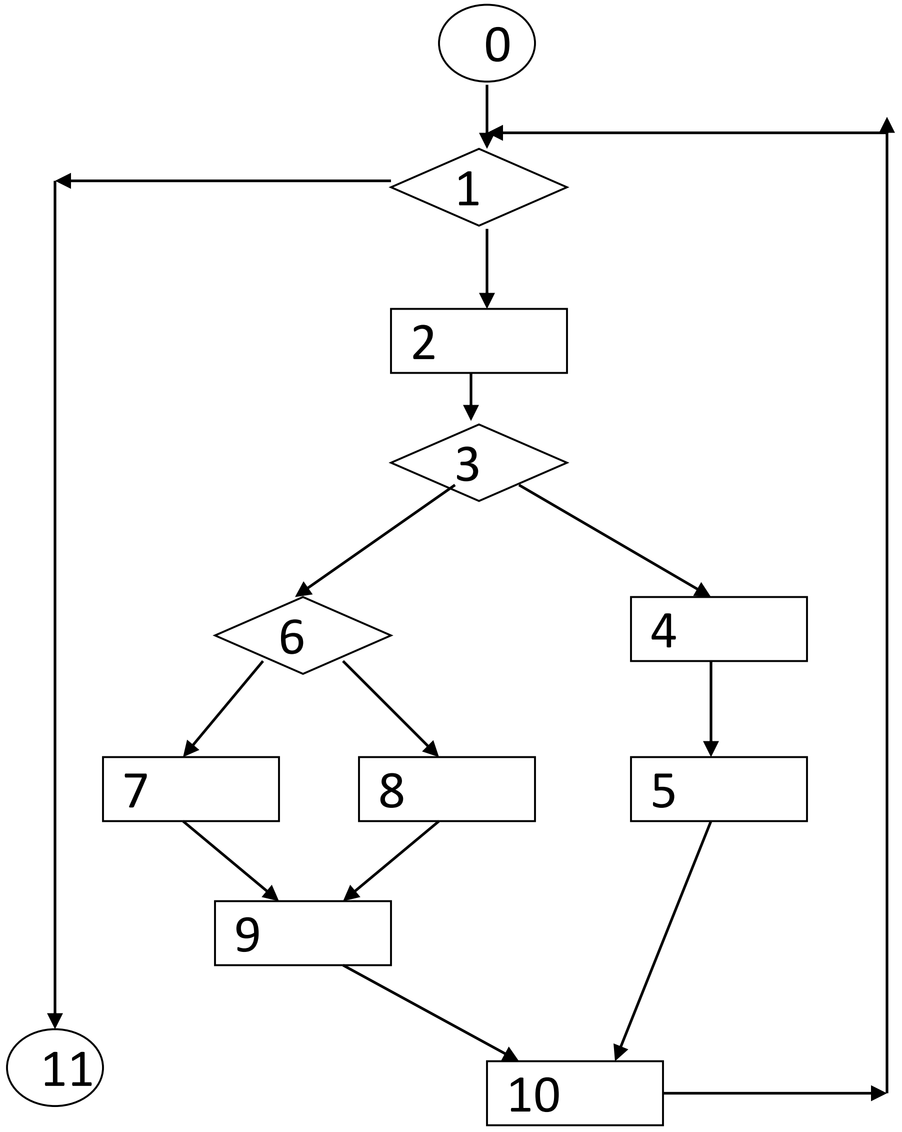
- White-box testing technique proposed by Tom McCabe
- Enables the test case designer to derive a logical complexity measure of a procedural design
- Uses this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise the basis set are guaranteed to execute <u>every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

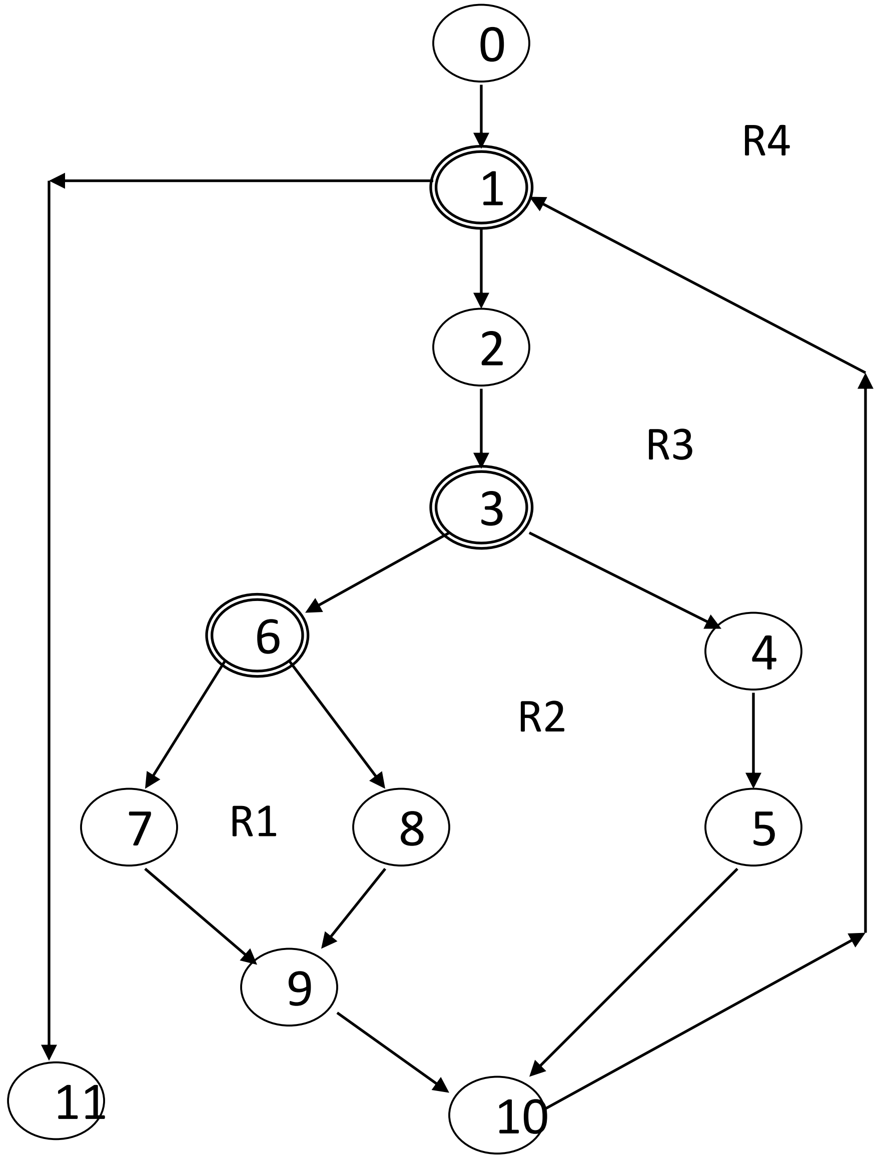
Flow Graph Notation

- 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

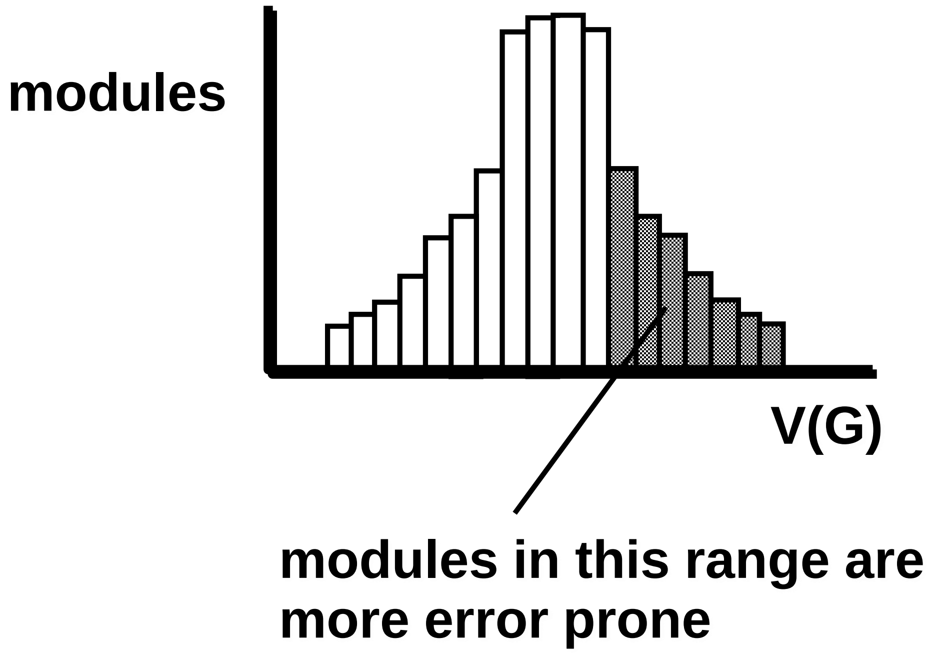
- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along <u>at least one</u> edge that has not been traversed before by a previous path
- Basis set for flow graph 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 <u>number of paths</u> in the basis set is determined by the <u>cyclomatic complexity</u>

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

Cyclomatic Complexity

A number of industry studies have indicated that higher the cyclomatic complexity, higher the probability or errors.



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

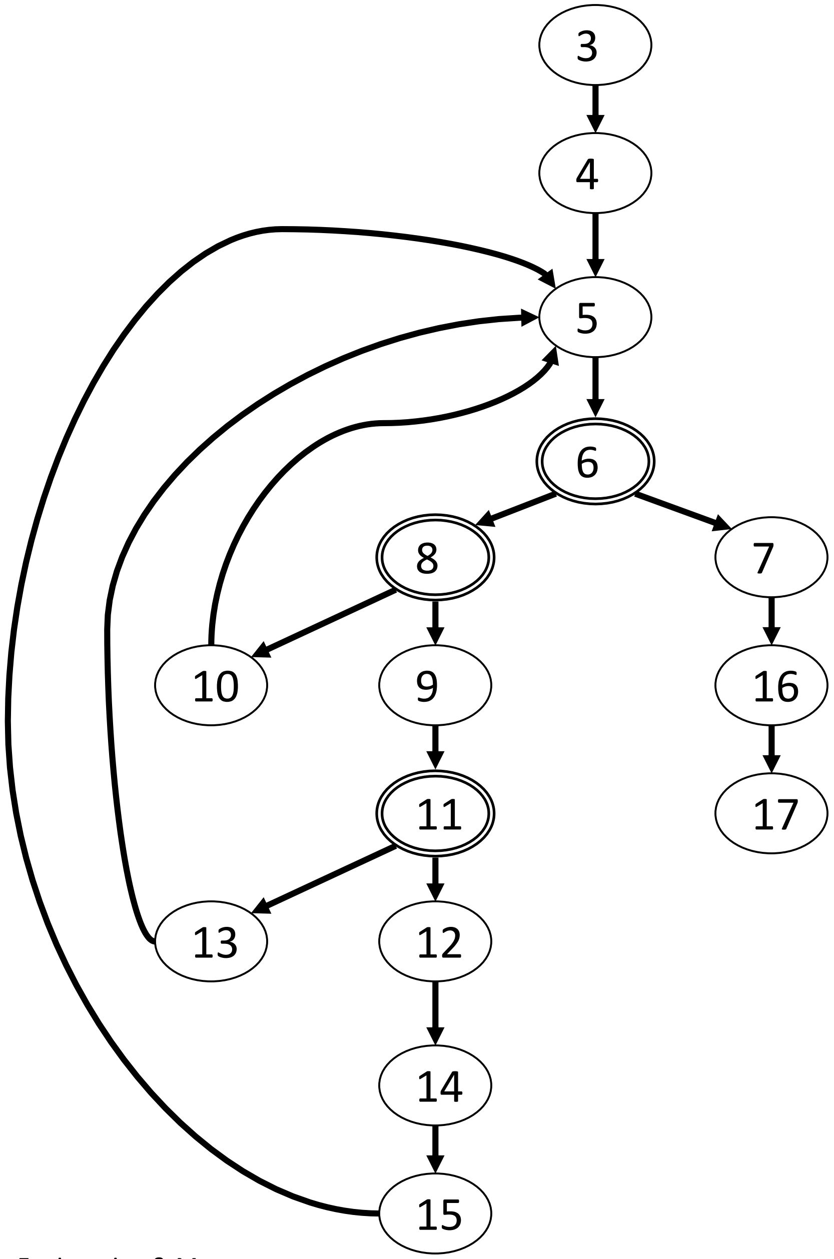
```
 1   int functionY(void)
 2   {
 3      int x = 0;
 4      int y = 19;

 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   }
```



Control Structure Testing

Condition testing — a test case design method that exercises the logical conditions contained in a program module

Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

- The data flow testing method (proposed by Frankl, Weiss [93]) selects test paths of a program according to the locations of definitions and uses of variables in the program.
  - Some statements "define" a variable’s value (i.e., a “variable definition”)
  - Some statements "use" variable’s value (i.e., a “variable use”)
  - A DU(definition-use) chain connects the statements and exposes anamolies

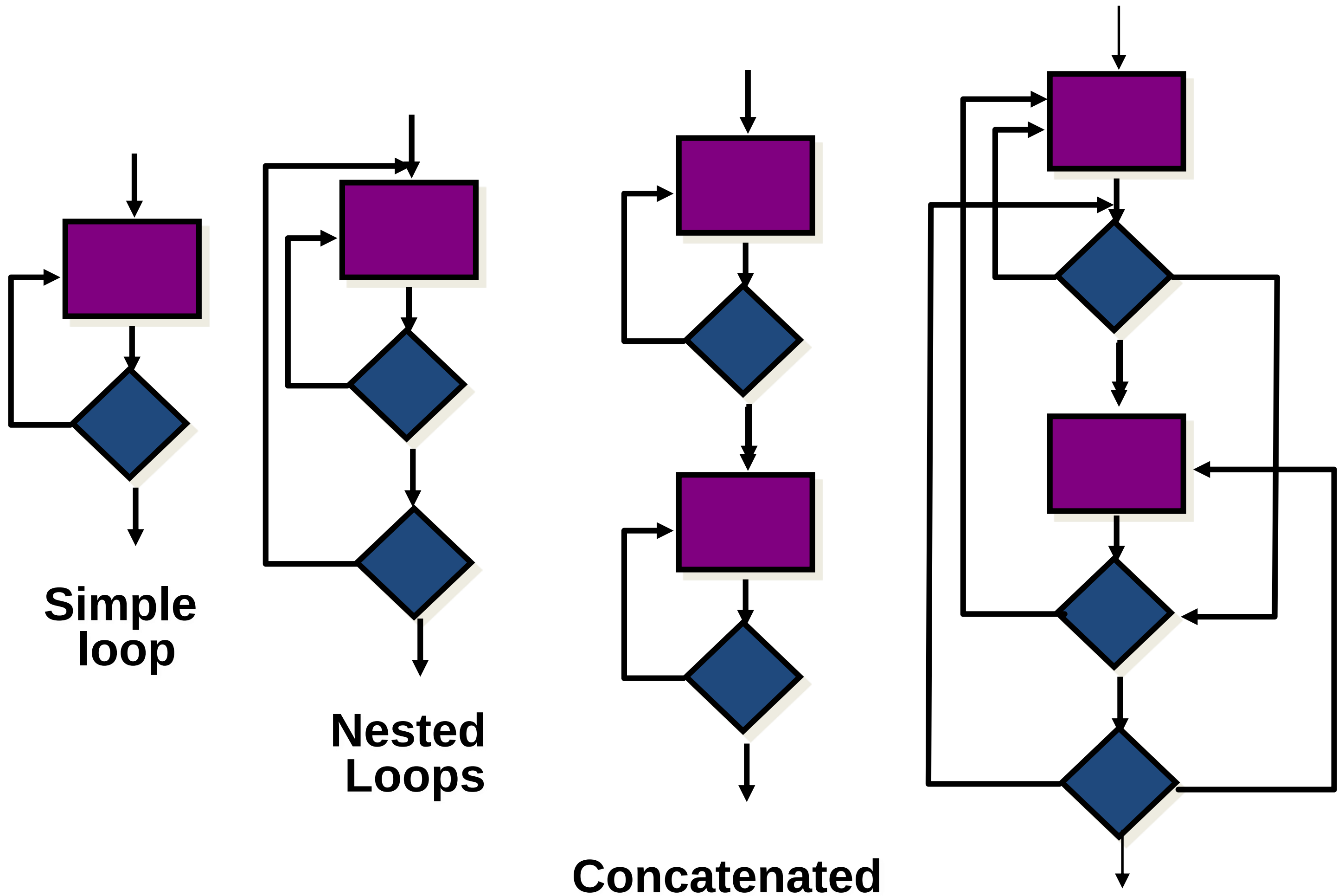
Loop Testing - General

- 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)
```

Loop Testing



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 <u>innermost</u> loop; set all other loops to <u>minimum</u> 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) <u>Work outward</u>, 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

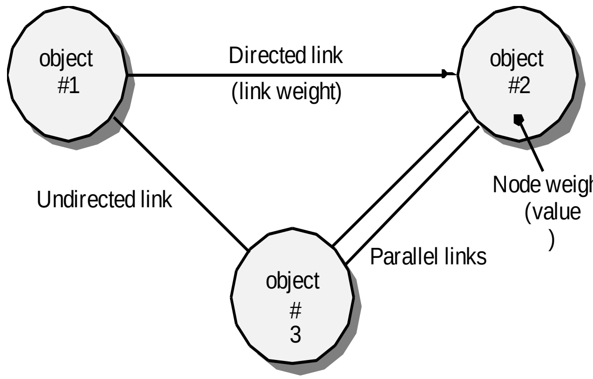
Questions answered by Black-box Testing

- 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?

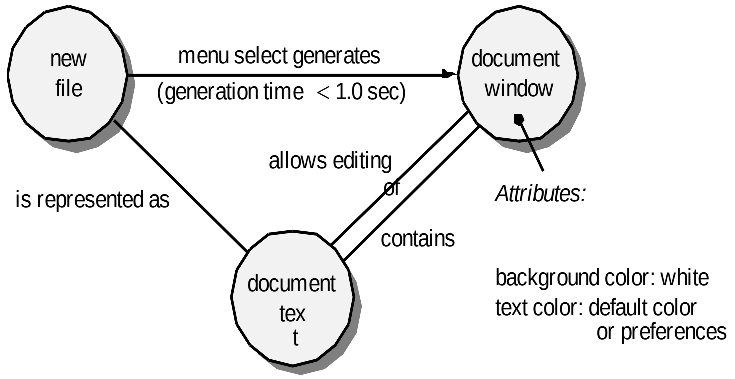
Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



(a)



(b)

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

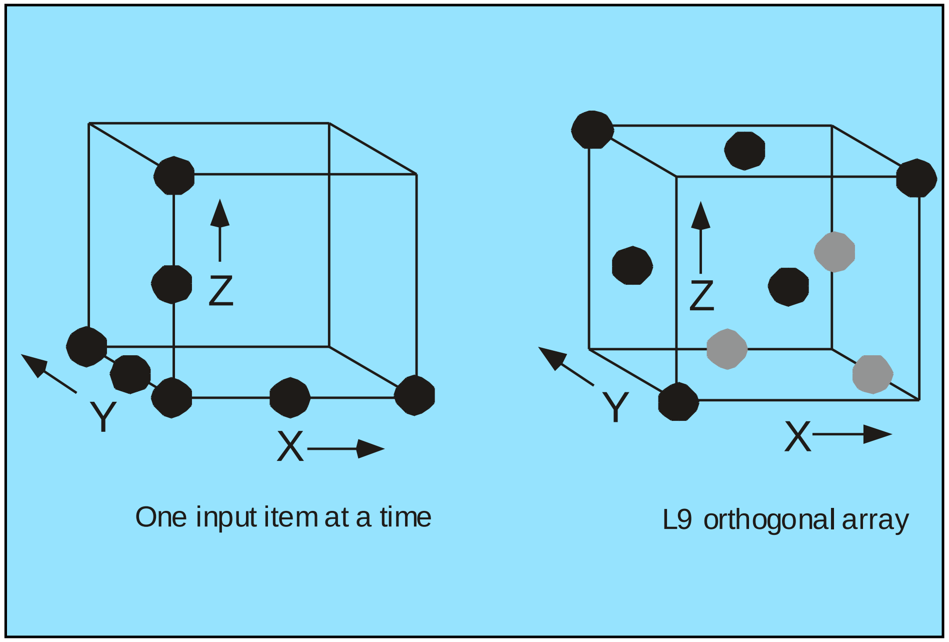
- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
  - It selects test cases at the edges of a class
  - It derives test cases from both the input domain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

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



	X	Y	Z
1	1	1	1
2	1	2	2
3	1	3	3
4	2	1	2
5	2	3	1
6	2	2	3
7	3	2	1
8	3	3	2
9	3	1	3

Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - Recall that a *behavioral model* indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.

Object-Oriented Testing Methods

OO Testing

To adequately test OO systems, three things must be done:

- the definition of testing must be broadened to include object-oriented analysis and design models
  - The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level
- the strategy for unit and integration testing must change significantly, and
- the design of test cases must account for the unique characteristics of OO software.

OO Testing Strategies

- Unit testing
  - the concept of the unit changes
  - the smallest testable unit is the encapsulated class
  - a single operation can no longer be tested in isolation (the conventional view of unit testing) but rather, as part of a class

- Integration Testing
  - Thread-based testing integrates the set of classes required to respond to one input or event for the system
  - Use-based testing begins the construction of the system by testing those classes (called independent classes) that use very few (if any) of server classes. After the independent classes are tested, the next layer of classes, called dependent classes
  - Cluster testing defines a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

- Validation Testing
  - details of class connections disappear
  - draw upon use cases that are part of the requirements model
  - Conventional black-box testing methods can be used to drive validation tests

Testing Focus

- Fault-based testing
  - 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.
- Class Testing and the Class Hierarchy
  - Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.
- Scenario-Based Test Design
  - Scenario-based testing 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 and their variants as tests.

OOT Methods

- Random testing
  - identify operations applicable to a class
  - define constraints on their use
  - generate a variety of random (but valid) test sequences to exercise more complex class instance life histories

- Partition Testing
  - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software
  - state-based partitioning
    - categorize and test operations based on their ability to change the state of a class
  - attribute-based partitioning
    - categorize and test operations based on the attributes that they use
  - category-based partitioning
    - categorize and test operations based on the generic function each performs

- Inter-class testing
  - For each client class, use the list of class operators to generate a series of random test sequences that will send messages to other server classes.

OOT Methods: Behavior Testing

The tests to be designed should achieve all state coverage [Kirani and Tsai - KIR94].

The operation sequences should cause the Account class to make transition through all allowable states

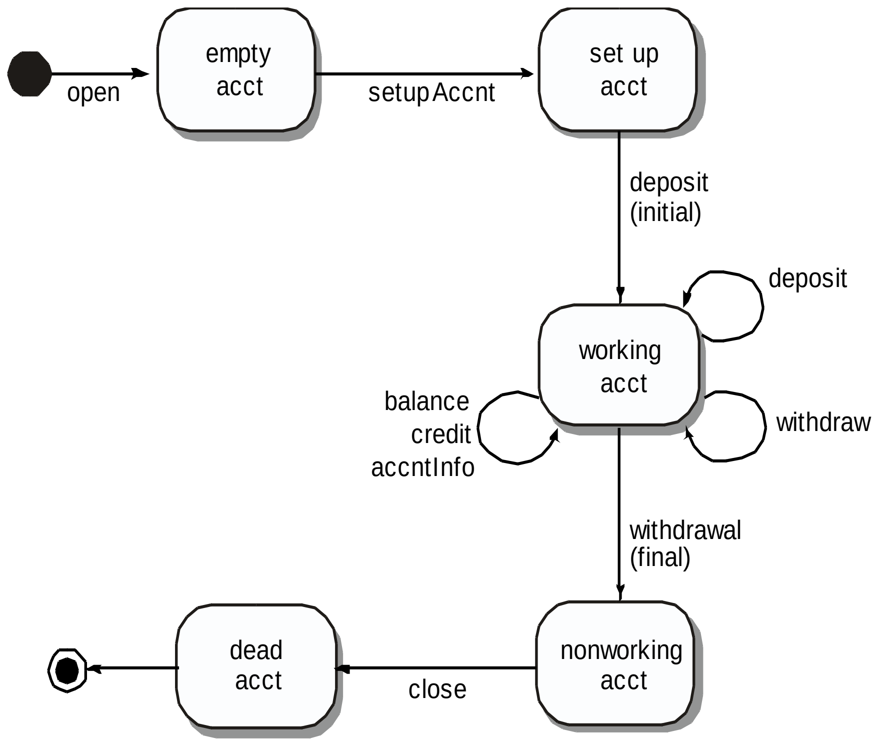


Figure 14.3 State diagram for Account class (adapted from [KIR94])

Software Testing Patterns

- Testing patterns are described in much the same way as design patterns
- *Example:*
    - *Pattern name:* **ScenarioTesting**
    - *Abstract:* Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The **ScenarioTesting** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement.

Manual vs Automated Testing

Manual Testing

- A manual tester has the ability to react to the context of a project, making it flexible and responsive to change. The ability to quickly adapt to change in particular suits a small business environment.
- In manual testing, it's possible to see the benefits of testing immediately, especially if the testing style is exploratory in nature and a tester is very quickly able to feed bugs and other information back to the rest of the team.
- Good testers require excellent analytical, problem-solving and communication skills combined with a healthy dose of skepticism.
- Including tools in testing can be a very powerful way of achieving added depth to your testing. A simple example would be the use of a tool to assist you in stress or load testing, something that a tester would struggle to test well manually.

How to Reduce the Cost of Software Testing by Matthew Heusser and Govind Kulkarni (eds) Auerbach Publications © 2012

Manual Testing

- Manual testing can be time-consuming; hence it may end up being expensive
- Since manual testing depends on individual tester, errors and discrepancies may arise
- Observations may not be consistently reported.
- Testers may find it boring to run simplistic test cases.
- Some intermittent errors (e.g. timing issues in synchronous programs) require many runs to detect; testers may not sustain necessary patience

Automated Testing

- Automating the parts of testing that humans do badly is a very effective way of improving the depth and scope of testing.

- What may take a manual tester days to test can often be exercised by automated scripts in minutes. It is advantageous to test large amounts of input data by automating the effort

- Automated testing can also be useful in repeating the tests. In some types of testing such as performance testing, automated tools are a necessity

- Reduces errors by removing humans from discrepancy detection task

How to Reduce the Cost of Software Testing by Matthew Heusser and Govind Kulkarni (eds) Auerbach Publications © 2012

Automated Testing

- Automation testing is entirely confirmatory in nature and will do little to exercise the application in any new way
- Automated testing requires up-front resources before testing is due to start. A test strategy and code need to be created upfront, ready for execution at the start of testing
- Also, like any code, automated scripts need to be maintained, potentially extending the cost of the testing.
- Organizations have to additionally invest in time to learn tools

How to Reduce the Cost of Software Testing by Matthew Heusser and Govind Kulkarni (eds) Auerbach Publications © 2012

Manual vs. Automated

- While automation evolves rapidly, manual testing is not replaceable.
- New features, complex validations, and business intensive functionalities continue to be tested manually
- Goal of 100% automation is impractical.
- Typically 70% automation helps maximize ROI.

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?

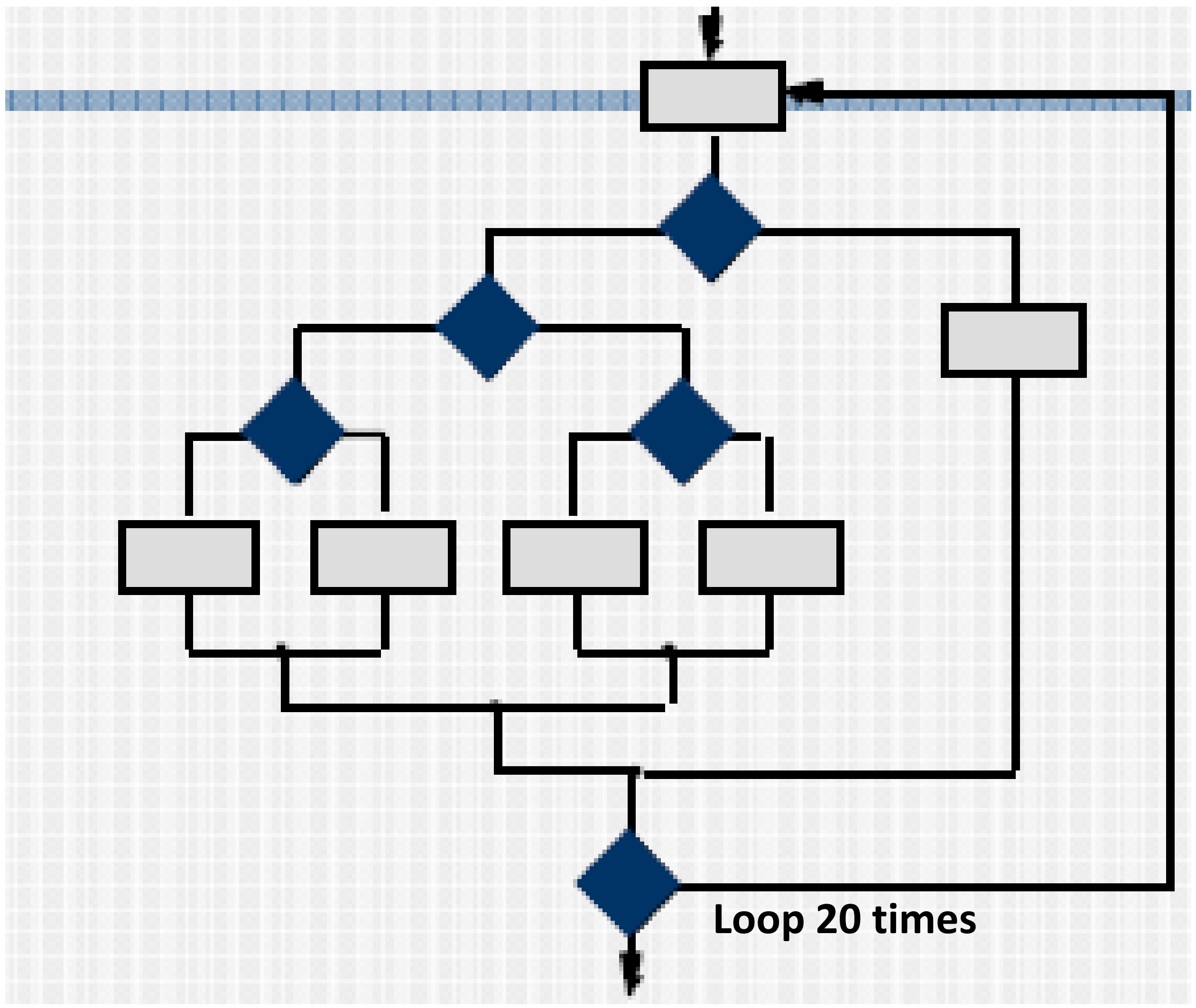
- The symptom and the cause may be <u>geographically remote</u>
- The symptom may <u>disappear (temporarily)</u> when another error is corrected
- The symptom may actually be caused by <u>nonerrors</u> (e.g., round-off accuracies)
- The symptom may be caused by <u>human error</u> that is not easily traced

(continued on next slide)

Why is Debugging so Difficult?
(continued)

- 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

Can you guess no of different execution paths?



Order of trillions!!!

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

Summary of Testing Methods

•Software engineer’s goal is to remove as many as possible early in the software development cycle.

•Testing can only find errors it cannot prove that a program is free of bugs.

•Two basic test techniques exist for testing coventional software, testing module input/output (black-box) and exercising the internal logic of software components (white-box).

•Test data is important for effectiveness of test case.

•Testing must be planned and designed.

•Debugging occurs as a consequence of successful testing

•It is still very much an art rather than a science

•The debugging process attempts to match symptom with cause, thereby leading to error correction