SOFTWARE TESTING STRATEGIES

22.1 A STRATEGIC APPROACH TO SOFTWARE TESTING

Explain a strategic approach to software testing.

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason, a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods—should be defi ned for the software process

22.1.1 Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of tasks that ensure that software correctly implements a specific function. Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way: Verification: "Are we building the product, right?" Validation: "Are we building the right product?" The definition of V&V encompasses many software quality assurance activities (Chapter 21).

1 Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary. Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective technical reviews, and solid management and measurement all lead to quality that is confirmed during testing. Miller [Mil77] relates software testing to quality assurance by stating that "the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems."

22.1.2 Organizing for Software Testing

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test it. This situation seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these developers have a vested interest in demonstrating that the program is error-free, works according to customer requirements, and will be completed on schedule and within budget. Each of these interests can mitigate against thorough testing.

From a psychological point of view, software analysis and design (along with coding) are constructive tasks. The software engineer analyses, models, and then creates a computer program and its documentation. Like any builder, the software engineer is proud of the edifice that has been built and may look askance at anyone who attempts to tear it down. When testing commences, there is a subtle, yet definite, attempt to "break" the thing that the software engineer has built. From the builder's perspective, testing can be considered psychologically destructive. Therefore, the builder might design and execute tests that demonstrate the program's functionality rather than uncover errors. Unfortunately, errors will still be present, and if the software engineer doesn't find them, the customer will.

Several misconceptions arise from this discussion:

1. The developer of software should do no testing at all.
2. The software should be "tossed over the wall" to strangers who will test it mercilessly.
3. Testers get involved with the project only when the testing steps are about to begin.

Each of these statements is incorrect. The software developer is always responsible for testing individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. In many cases, the developer also conducts integration testing—a step that leads to the construction and testing of the complete software architecture. Only after the software architecture is complete does an independent test group (ITG) become involved.

The role of an ITG is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing helps to remove the conflict of interest that might otherwise be present, as ITG personnel are specifically tasked with finding errors.

However, you don't simply turn the program over to the ITG and walk away. The developer and the ITG work closely throughout the project to ensure that thorough tests will be conducted. During testing, the developer must be available to correct errors that are uncovered.

The ITG is considered part of the software development project team, as it becomes involved during analysis and design and stays involved (planning and specifying test procedures) throughout the project. In many cases, the ITG reports to the software quality assurance organization, achieving a degree of independence that might not be possible if it were part of the software engineering team.

22.1.3 Software Testing Strategy—The Big Picture

Figure 22.1 Testing strategy

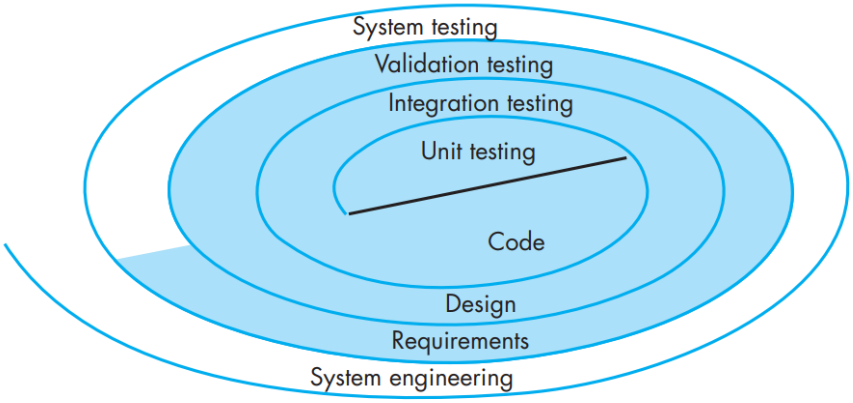
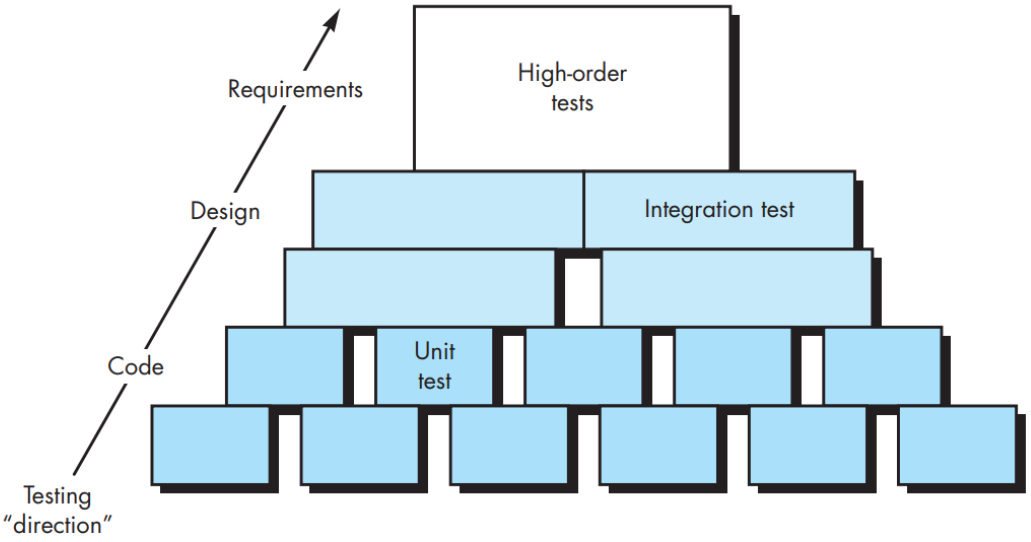


Figure 22.2 Software testing steps



The software process may be viewed as the spiral illustrated in Figure 22.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behaviour, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward along streamlines that decrease the level of abstraction on each turn.

A strategy for software testing may also be viewed in the context of the spiral (Figure 22.1).

Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter validation testing, where requirements established as part of requirements modelling are validated against the software that has been constructed. Finally, you arrive at system testing, where the software and other system elements are tested

as a whole. To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure 22.2. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name unit testing. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.

Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Test-case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.

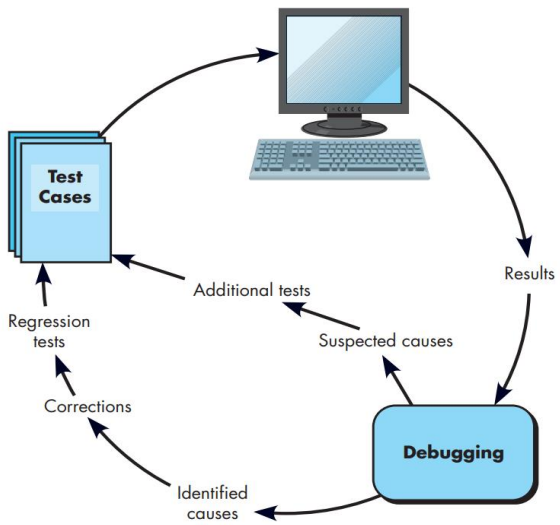
After the software has been integrated (constructed), a set of high-order tests is conducted. Validation criteria (established during requirements analysis) must be evaluated. Validation testing provides final assurance that software meets all functional, behavioural, and performance requirements.

22.9 THE ART OF DEBUGGING

Software testing is a systematic process involving planned test-case design, defined strategies, and evaluation against expectations. Debugging, which follows successful testing, is the process of removing errors uncovered by tests. While debugging can be orderly, it remains art. Software engineers often face "symptomatic" indications of problems, where the error's external manifestation and its internal cause may not be obviously related. The mental process connecting a symptom to its cause is the essence of debugging

22.9.1 The Debugging Process **With a neat diagram, describe the debugging process.**

FIGURE 22.7
The debugging process



Debugging is not testing but often occurs as a consequence of testing. 6 Referring to Figure 22.7, the debugging process begins with the execution of a test case.

Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction. The debugging process will usually have one of two outcomes:

(1) the cause will be found and corrected or

(2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion. Why is debugging so difficult? In all likelihood, human psychology (see Section 22.9.2) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components (Chapter 12) exacerbate this situation.

2. The symptom may disappear (temporarily) when another error is corrected.

3. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).

4. The symptom may be caused by human error that is not easily traced.

5. The symptom may be a result of timing problems, rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

8. The symptom may be due to causes that are distributed across a number of tasks running on different processors. During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure forces a software developer to fix one error and at the same time introduce two more.

Compare and contrast top-down and bottom-up integration testing strategies.

Characteristic	Top-Down Integration Testing	Bottom-Up Integration Testing
Approach	Tests the highest-level modules first and then works down the hierarchy.	Tests the lowest-level modules first and then works up the order.
Stubs	It uses stubs to simulate the lower-level modules.	It uses drivers to simulate the higher-level modules.
Complexity	Less complex.	More complex.
Data Intensity	Less data intensive.	More data intensive.
Risk coverage	Focuses on identifying and mitigating risks early on.	Focuses on identifying and mitigating risks later in the testing process.
Scope	It can be used to test large and complex systems.	It is better suited for testing smaller and less complex systems.
Suitability	It is suitable for systems that have a clear hierarchical structure.	It is suitable for systems that need a clear hierarchical structure.
Benefits	It can help to identify and mitigate risks early on.	It can help to ensure that the lower-level modules are working as expected.
Drawbacks	It can be challenging to implement for large and complex systems.	It can be time-consuming and data-intensive to execute.
Examples	Operating systems, database systems, and word-processing software.	Device drivers, embedded systems, and microcontrollers.
Other Differences	Main module calls the sub modules Uses stubs as a replacement for missing sub modules Implemented on structured/procedure-oriented programming Significant to identify errors at the top levels Difficult to observe the output Simple to perform	Sub modules integrated with the top module(s) Makes use of drivers as a replacement for main/top modules Beneficial for object-oriented programming Good to determine defects at lower levels Easy to observe and record the results Highly complex and data-driven

Will exhaustive testing guarantee that the program is 100 percent correct? Illustrate with suitable examples

Exhaustive Testing:

Consider a 100-line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately 10 power 14 possible paths that may be executed in this program! To put this number in perspective, we assume that a magic test

processor (“magic” because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause havoc in most development schedules. Therefore, it is reasonable to assert that exhaustive testing is impossible for large software systems.

Will exhaustive testing guarantee that the program is 100 percent correct? Illustrate with suitable examples.

No, even exhaustive testing will not guarantee that the program is 100 percent correct. There are too many variables to consider. Consider this...

Installation testing - did the program install according to the instructions? Integration testing - did the program work with all of the other programs on the system without interference, and did the installed modules of the program integrate and work with other installed modules?

Function testing - did each of the program functions work properly? Unit testing - did the unit work as a standalone as designed, and did the unit work when placed in the overall process? User Acceptance Testing - did the program fulfil all of the user requirements and work per the user design? Performance testing - did the program perform to a level that was satisfactory and could it carry the volume load placed upon it? While these are just the basic tests for an exhaustive testing scenario, you could keep testing beyond these tests using destructive methods, white box internal program testing, establish program exercises using automated scripts, etc. The bottom line is... testing has to stop at some point in time. Either the time runs out that was allotted for testing, or you gain a confidence level that the program is going to work. (Of course, the more you test, the higher your confidence level). I don't know anyone that would give a 100% confidence level that the program is 100% correct, (to do so is to invite people to prove you wrong and they will come back with all kinds of bugs you never even considered). However, you may be 95% confident that you found most all of the major bugs. Based upon this level of confidence, you would then place the program into production use - always expecting some unknown bug to be found.

Difference between Black Box Testing and White Box Testing:

Parameters	Black Box Testing	White Box Testing
Definition	Black Box Testing is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	White Box Testing is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
Testing objectives	Black box testing is mainly focused on testing the functionality of the software, ensuring that it meets the requirements and specifications.	White box testing is mainly focused on ensuring that the internal code of the software is correct and efficient.
Testing methods	Black box testing uses methods like equivalence partitioning, boundary value analysis, and error guessing to create test cases.	White box testing uses methods like control flow testing, data flow testing and statement coverage testing.
Knowledge level	Black box testing does not require any knowledge of the internal workings of the software, and can be performed by testers who are not familiar with programming languages.	White box testing requires knowledge of programming languages, software architecture and design patterns.
Scope	Black box testing is generally used for testing the software at the functional level.	White box testing is used for testing the software at the unit level, integration level and system level.
Implementation	Implementation of code is not needed for black box testing.	Code implementation is necessary for white box testing.

Parameters	Black Box Testing	White Box Testing
Done By	Black Box Testing is mostly done by software testers.	White Box Testing is mostly done by software developers.
Terminology	Black Box Testing can be referred to as outer or external software testing.	White Box Testing is the inner or the internal software testing.
Testing Level	Black Box Testing is a functional test of the software.	White Box Testing is a structural test of the software.
Testing Initiation	Black Box testing can be initiated based on the requirement specifications document.	White Box testing of software is started after a detail design document.
Programming	No knowledge of programming is required.	It is mandatory to have knowledge of programming.
Testing Focus	Black Box Testing is the behavior testing of the software.	White Box Testing is the logic testing of the software.
Applicability	Black Box Testing is applicable to the higher levels of testing of software.	White Box Testing is generally applicable to the lower levels of software testing.
Alternative Names	Black Box Testing is also called closed testing.	White Box Testing is also called as clear box testing.
Time Consumption	Black Box Testing is least time consuming.	White Box Testing is most time consuming.

Parameters	Black Box Testing	White Box Testing
Suitablefor Algorithm Testing	Black Box Testing is not suitable or preferred for algorithm testing.	White Box Testing is suitable for algorithm testing.
Approach	Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
Example	Search something on google by using keywords	By input to check and verify loops
Exhaustiveness	It is less exhaustive as compared to white box testing.	It is comparatively more exhaustive than black box testing.
Types	**Types of Black Box Testing:** • Functional Testing • Non-functional testing • Regression Testing	**Types of White Box Testing:** • Unit Testing • Integration Testing • Regression Testing

Design various test cases for unified seat reservation system.

1. User Authentication

Test Case 1: User Login

- **Precondition:** User is registered in the system.
- **Action:** User tries to log in with valid credentials.
- **Expected Result:** User successfully logs in and is redirected to the home page.

Test Case 2: User Login with Invalid Credentials

- **Precondition:** User is registered in the system.
- **Action:** User tries to log in with invalid credentials.

- **Expected Result:** User receives an error message indicating incorrect credentials.

Test Case 3: Password Recovery

- **Precondition:** User is registered in the system.
- **Action:** User requests a password recovery.
- **Expected Result:** User receives an email with password recovery instructions.

2. Seat Availability and Selection

Test Case 4: Check Seat Availability

- **Precondition:** Multiple seats are available in the system.
- **Action:** User checks seat availability for a specific date and time.
- **Expected Result:** The system displays available seats for the selected date and time.

Test Case 5: Select a Seat

- **Precondition:** Seats are available for the chosen event.
- **Action:** User selects a specific seat.
- **Expected Result:** The selected seat is marked as reserved and is no longer available for other users.

Test Case 6: Select an Already Reserved Seat

- **Precondition:** A seat is already reserved by another user.
- **Action:** User tries to select the already reserved seat.
- **Expected Result:** The system shows an error message indicating the seat is already reserved.

3. Reservation Confirmation and Payment

Test Case 7: Confirm Reservation

- **Precondition:** User has selected a seat.
- **Action:** User confirms the reservation.
- **Expected Result:** Reservation details are saved, and a confirmation message is displayed.

Test Case 8: Make Payment

- **Precondition:** User has confirmed a reservation.
- **Action:** User makes a payment using a valid payment method.
- **Expected Result:** Payment is processed successfully, and a receipt is generated.

Test Case 9: Payment Failure

- **Precondition:** User has confirmed a reservation.
- **Action:** User makes a payment using an invalid payment method.
- **Expected Result:** Payment fails, and the system shows an appropriate error message.

4. Cancellation and Modification

Test Case 10: Cancel Reservation

- **Precondition:** User has a confirmed reservation.
- **Action:** User cancels the reservation.
- **Expected Result:** Reservation is cancelled, and the seat is made available again.

Test Case 11: Modify Reservation

- **Precondition:** User has a confirmed reservation.
- **Action:** User modifies the reservation (e.g., changes the seat or date).
- **Expected Result:** The reservation details are updated accordingly.

5. System Performance and Reliability

Test Case 12: Load Test

- **Precondition:** System is running under normal conditions.
- **Action:** Simulate multiple users trying to access the system simultaneously.
- **Expected Result:** The system handles concurrent users without performance degradation or crashes.

Test Case 13: Stress Test

- **Precondition:** System is running under normal conditions.
- **Action:** Push the system beyond its maximum capacity.
- **Expected Result:** The system gracefully handles the overload and recovers without data loss.

6. Security

Test Case 14: SQL Injection

- **Precondition:** System has input fields.
- **Action:** User attempts an SQL injection attack.
- **Expected Result:** The system rejects the input and prevents unauthorized database access.

Test Case 15: Cross-Site Scripting (XSS)

- **Precondition:** System has input fields.
- **Action:** User attempts to inject malicious scripts.
- **Expected Result:** The system sanitizes inputs and prevents the execution of malicious scripts.

7. Usability

Test Case 16: User Interface Navigation

- **Precondition:** User is logged in.
- **Action:** User navigates through different sections of the application.
- **Expected Result:** Navigation is smooth, and all elements are accessible and functioning as expected.

Test Case 17: Accessibility

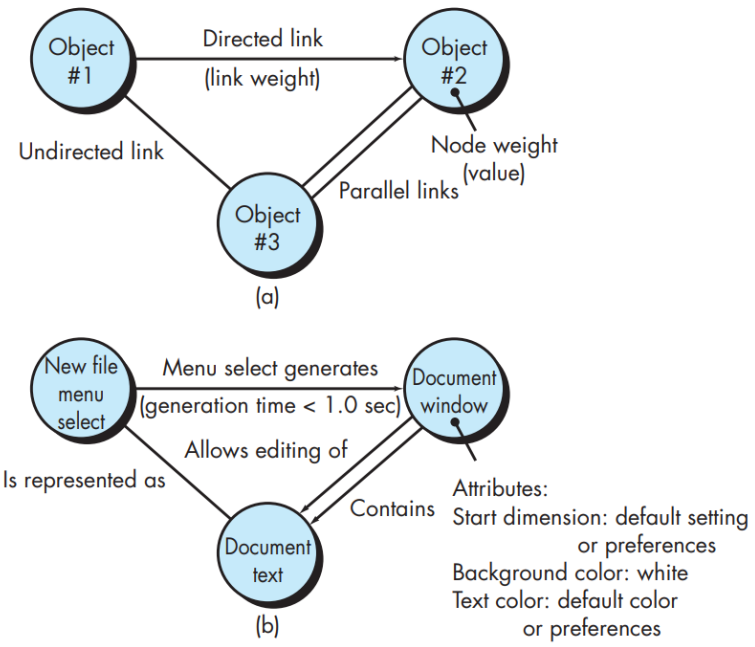
- **Precondition:** User with disabilities tries to use the system.
- **Action:** User uses screen readers or keyboard navigation.
- **Expected Result:** The system is fully accessible, meeting WCAG (Web Content Accessibility Guidelines) standards.

Explain graph-based testing methods and boundary value analysis with suitable real time examples.

23.6.1 Graph-Based Testing Methods

FIGURE 23.8

(a) Graph notation; (b) simple example



The first step in black-box testing is to understand the objects that are modelled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another" [Bei95]. In other words, software testing begins by creating a graph of important objects and their relationships. Then, a series of tests are devised to cover the graph so that each object and relationship is exercised, and errors are uncovered.

To accomplish these steps, you begin by creating a graph—a collection of nodes that represent objects, links that represent the relationships between objects, node weights that describe the properties of a node (e.g., a specific data value or state behaviour), and link weights that describe some characteristic of a link. The symbolic representation of a graph is shown in Figure 23.8a. Nodes are represented as circles connected by links that take different forms. A directed link (represented by an arrow) indicates that a relationship moves in only one direction. A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 23.8b) where:

- **Object #1:** newFile (menu selection)
- **Object #2:** documentWindow
- **Object #3:** documentText

Referring to the figure, a menu selection on newFile generates a document window. The node weight of documentWindow provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the newFile menu selection and documentText, and parallel links indicate relationships between documentWindow and documentText. In reality, a far more detailed graph would have to be generated as a precursor to test-case design. You can then derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer [Bei95] describes a number of behavioral testing methods that can make use of graphs:

- **Transaction Flow Modelling:** The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps. For example, a data object flight Information Input is followed by the operation validation Availability Processing().
- **Finite State Modelling:** The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent

the transitions that occur to move from state to state (e.g., orderInformation is verified during inventoryAvailabilityLook-up() and is followed by customerBillingInformation input). The state diagram (Chapter 11) can be used to assist in creating graphs of this type.

- **Data Flow Modeling:** The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node FICATaxWithheld (FTW) is computed from gross wages (GW) using the relationship, FTW = 0.62 × GW.
- **Timing Modeling:** The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

23.6.3 Boundary Value Analysis

A greater number of errors occur at the boundaries of the input domain rather than in the "center." It is for this reason that boundary value analysis (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [Mye79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values aaa and bbb, test cases should be designed with values aaa and bbb and just above and just below aaa and bbb.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below the minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Real-Time Examples of Boundary Value Analysis

1. Login System:

- **Scenario:** A login system requires a username and a password. The username must be between 5 to 15 characters long, and the password must be between 8 to 20 characters long.
- **Boundary Test Cases:**
  - Username: 4, 5, 6, 14, 15, 16 characters long.
  - Password: 7, 8, 9, 19, 20, 21 characters long.

2. Online Shopping Cart:

- **Scenario:** An online shopping cart allows a user to add between 1 and 50 items.
- **Boundary Test Cases:**
  - Number of items: 0, 1, 2, 49, 50, 51.

3. Age Input for Registration Form:

- **Scenario:** A registration form requires the user to enter their age, which must be between 18 and 65.
- **Boundary Test Cases:**
    - Age: 17, 18, 19, 64, 65, 66.

4. Bank Transaction Limits:

- **Scenario:** A bank allows daily transactions between $10 and $10,000.
- **Boundary Test Cases:**
    - Transaction amount: $9, $10, $11, $9,999, $10,000, $10,001.

5. File Upload Size Limit:

- **Scenario:** A web application allows file uploads with a size limit between 1 MB and 100 MB.
- **Boundary Test Cases:**
    - File size: 0.9 MB, 1 MB, 1.1 MB, 99 MB, 100 MB, 100.1 MB.

6. Grading System:

- **Scenario:** A grading system assigns grades based on scores: A for 90-100, B for 80-89, C for 70-79, D for 60-69, and F for 0-59.
- **Boundary Test Cases:**
    - Scores: -1, 0, 1, 59, 60, 61, 69, 70, 71, 79, 80, 81, 89, 90, 91, 100, 101.

7. Temperature Monitoring System:

- **Scenario:** A temperature monitoring system should operate within the range of -20°C to 50°C.
- **Boundary Test Cases:**
    - Temperature: -21°C, -20°C, -19°C, 49°C, 50°C, 51°C.

8. Password Strength Validator:

- **Scenario:** A password must contain at least one uppercase letter, one lowercase letter, one digit, and be between 8 to 16 characters long.
- **Boundary Test Cases:**
    - Password length:

Online Shopping Cart Example with Boundary Test Cases

Scenario:

An online shopping cart allows a user to add between 1 and 50 items.

Boundary Test Cases:

1. **Below Minimum Boundary:**
    - **Number of items:** 0
    - **Expected Result:** Error message or validation failure indicating that at least one item must be added.
2. **Minimum Boundary:**
    - **Number of items:** 1
    - **Expected Result:** Successfully add 1 item to the cart.
3. **Just Above Minimum Boundary:**

- **Number of items:** 2
- **Expected Result:** Successfully add 2 items to the cart.

4. **Just Below Maximum Boundary:**
   - **Number of items:** 49
   - **Expected Result:** Successfully add 49 items to the cart.
5. **Maximum Boundary:**
   - **Number of items:** 50
   - **Expected Result:** Successfully add 50 items to the cart.
6. **Above Maximum Boundary:**
   - **Number of items:** 51
   - **Expected Result:** Error message or validation failure indicating that the maximum limit of 50 items has been exceeded.

23.4 BASIS PATH TESTING

Explain Basis path testing in detail

With suitable example, explain basis path testing in detail.

With suitable example, explain basis path testing in detail.

Basis Path Testing Overview

Basis path testing is a white-box testing technique developed by Tom McCabe. It helps test-case designers derive a logical complexity measure for a procedural design and use this measure to define a basis set of execution paths. Test cases created from the basis set ensure every program statement executes at least once during testing.

23.4.1 Flow Graph Notation

Before using the basis path method, a simple notation for representing control flow, known as a flow graph (or program graph), is introduced. The flow graph illustrates logical control flow using symbols for structured constructs.

Flow Graph Example

FIGURE 23.1
Flow graph notation

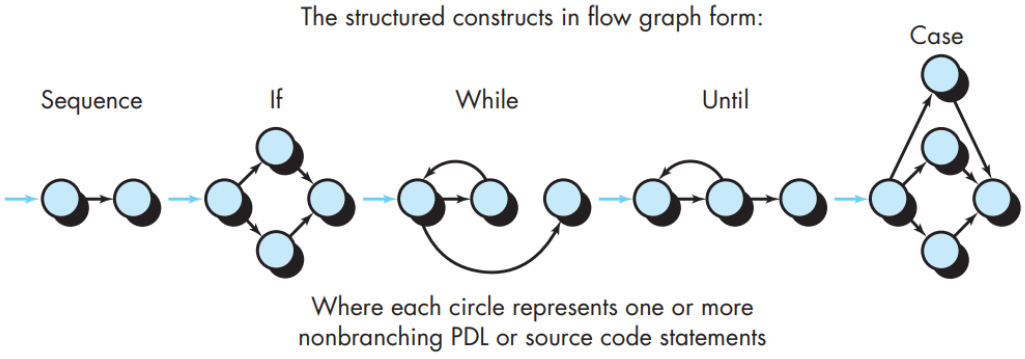
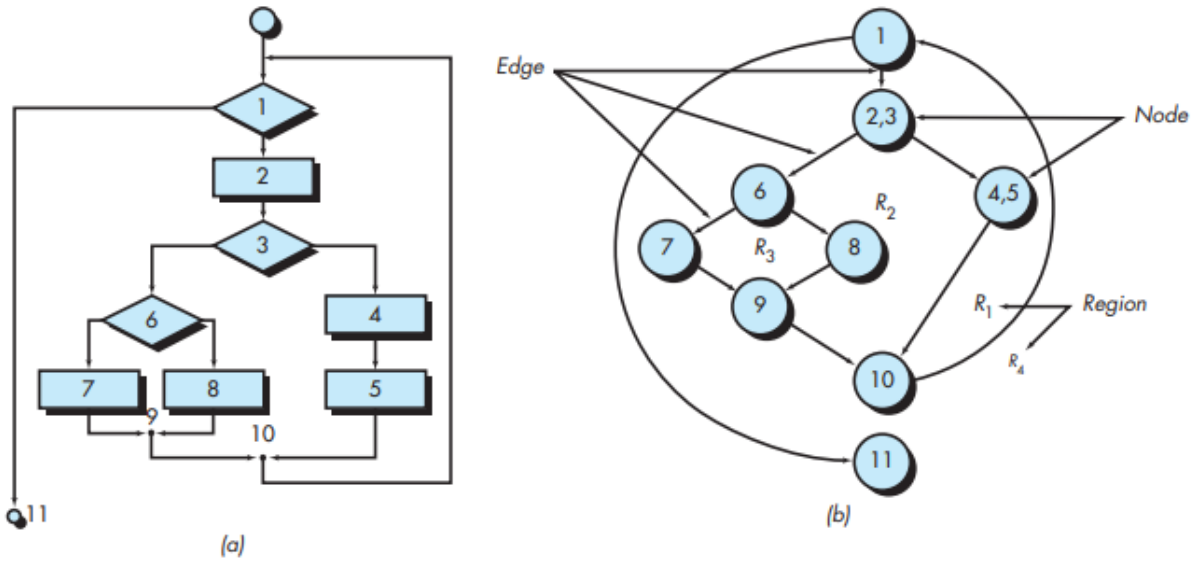


FIGURE 23.2 (a) Flowchart and (b) flow graph



- A flowchart in Figure 23.2a can be converted into a flow graph (Figure 23.2b).
- Each circle in the flow graph (node) represents one or more procedural statements.
- Arrows in the flow graph (edges or links) represent control flow.
- Nodes that contain conditions are called predicate nodes, characterized by two or more edges.

23.4.2 Independent Program Paths

An **independent path** introduces at least one new set of processing statements or a new condition. In terms of a flow graph, it moves along at least one edge not previously traversed. For example:

- **Paths for Figure 23.2b**:
  - Path 1: 1-11
  - Path 2: 1-2-3-4-5-10-1-11
  - Path 3: 1-2-3-6-8-9-10-1-11
  - Path 4: 1-2-3-6-7-9-10-1-11

Cyclomatic Complexity helps determine the number of independent paths:

- It's computed in three ways:
  1. Number of regions in the flow graph.
  2. V(G)=E−N+2V(G) = E - N + 2V(G)=E−N+2 (E = edges, N = nodes)
  3. V(G)=P+1V(G) = P + 1V(G)=P+1 (P = predicate nodes)

23.4.3 Deriving Test Cases

Steps to apply the basis path testing method:

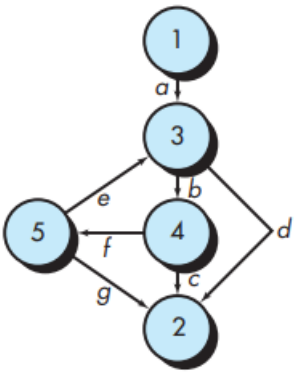
1. **Draw a Flow Graph**: Use design or code to create a flow graph.
2. **Determine Cyclomatic Complexity**: Compute V(G)V(G)V(G).
3. **Determine Basis Set of Paths**: Specify paths based on V(G)V(G)V(G).
4. **Prepare Test Cases**: Design tests to execute each path in the basis set.

23.4.4 Graph Matrices

A **graph matrix** is a data structure useful for developing software tools to assist in basis path testing. It's a square matrix where each row and column correspond to a node in the flow graph, and entries represent

connections (edges) between nodes. Additional properties (link weights) like execution probability, processing time, memory, or resources can be assigned to edges for enhanced analysis.

FIGURE 23.6
Graph matrix



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Basis Path Testing Explained with Example

Basis path testing is a white-box testing technique used to derive a logical complexity measure of a procedural design and use this measure to define a basis set of execution paths. Test cases created from this basis set ensure that every statement in the program is executed at least once during testing.

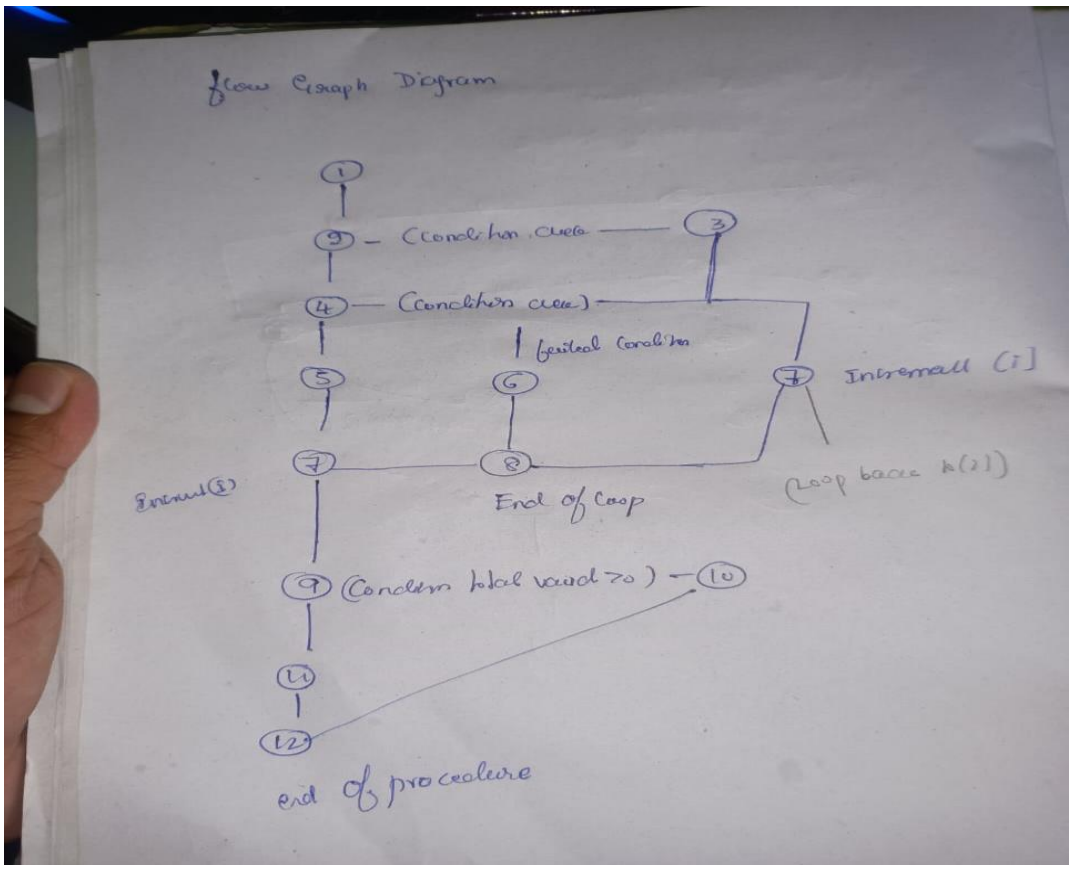
Steps for Basis Path Testing

1. **Draw the Flow Graph**
2. **Determine Cyclomatic Complexity**
3. **Determine Basis Set of Paths**
4. **Prepare Test Cases**

Example: Procedure for Calculating Average

Consider a simple procedure average to compute the average of 100 or fewer numbers that lie between bounding values. This procedure also computes the sum and total number of valid inputs.

Step 1: Draw the Flow Graph



The flow graph for the average procedure is created by mapping the PDL statements into flow graph nodes.

Step 2: Determine Cyclomatic Complexity

Cyclomatic complexity V(G)V(G)V(G) is determined using one of the three methods:

1. **Regions Method**:
    - Number of regions = 6
2. **Edges and Nodes Method**:
    - V(G)=E−N+2V(G) = E - N + 2V(G)=E−N+2
    - E=17E = 17E=17 (number of edges)
    - N=13N = 13N=13 (number of nodes)
    - V(G)=17−13+2=6V(G) = 17 - 13 + 2 = 6V(G)=17−13+2=6
3. **Predicate Nodes Method**:
    - V(G)=P+1V(G) = P + 1V(G)=P+1
    - P=5P = 5P=5 (number of predicate nodes)
    - V(G)=5+1=6V(G) = 5 + 1 = 6V(G)=5+1=6

Thus, V(G)V(G)V(G) for the average procedure is 6.

Step 3: Determine Basis Set of Paths

Based on the cyclomatic complexity, we identify 6 independent paths:

1. **Path 1**: 1-2-10-11-13
2. **Path 2**: 1-2-10-12-13
3. **Path 3**: 1-2-3-10-11-13
4. **Path 4**: 1-2-3-4-5-8-9-2-... (loop)
5. **Path 5**: 1-2-3-4-5-6-8-9-2-... (loop)
6. **Path 6**: 1-2-3-4-5-6-7-8-9-2-... (loop)

The ellipsis (...) indicates that the paths can loop back through the control structure until the end condition is met.

Step 4: Prepare Test Cases

For each independent path, create test cases to ensure all conditions and statements are executed:

1. **Test Case 1**: Input values that skip the loop entirely.
    - Values: [-999]
    - Expected Result: average = -999
2. **Test Case 2**: Input values that traverse the loop but do not meet the valid condition.
    - Values: [0, -999], minimum: 1, maximum: 100
    - Expected Result: average = -999
3. **Test Case 3**: Input values that traverse the loop and meet the valid condition.
    - Values: [50, -999], minimum: 1, maximum: 100
    - Expected Result: average = 50
4. **Test Case 4**: Input values that loop and meet the valid condition intermittently.
    - Values: [50, 150, -999], minimum: 1, maximum: 100
    - Expected Result: average = 50
5. **Test Case 5**: Input values that traverse and meet the valid condition more frequently.
    - Values: [50, 75, -999], minimum: 1, maximum: 100
    - Expected Result: average = 62.5
6. **Test Case 6**: Input values that loop until the maximum count of 100.
    - Values: [repeated valid values up to 100 times]
    - Expected Result: Correct computation of average

Describe any three system testing types with real time examples.

System testing is a critical phase in the software testing lifecycle that focuses on evaluating the complete and integrated software system to ensure it meets the specified requirements. Here are three common types of system testing, along with real-time examples:

1. Functional Testing

Description: Functional testing verifies that the software behaves according to the defined requirements. It focuses on the functionality of the system, ensuring that all features work as expected.

Example: Online Banking System

For an online banking application, functional testing would involve verifying:

- **Account Management**: Users can create, view, update, and delete their accounts.
- **Transaction Processing**: Users can perform transfers between accounts, pay bills, and view transaction history.
- **Login and Authentication**: The system properly authenticates users based on their credentials and provides appropriate access to functionalities.

Test Scenario: Testing the "Transfer Funds" feature to ensure that users can successfully transfer money from one account to another, that the transaction is accurately reflected in both accounts, and that appropriate notifications are sent.

2. Performance Testing

Description: Performance testing assesses the system's responsiveness, stability, and scalability under various conditions. It aims to ensure that the software performs well under expected and peak loads.

Example: E-commerce Website During Black Friday Sale

For an e-commerce website, performance testing would involve:

- **Load Testing**: Simulating a large number of simultaneous users to ensure the website can handle high traffic during peak shopping times.
- **Stress Testing**: Testing the website's behavior under extreme conditions, such as handling an unexpected surge in traffic or transaction volume.
- **Scalability Testing**: Assessing how well the website scales with increased user load or data volume, such as adding more servers or optimizing databases.

Test Scenario: Simulating thousands of users accessing the website simultaneously during a Black Friday sale to verify that the system can handle the load without significant slowdowns or crashes.

3. Security Testing

Description: Security testing identifies vulnerabilities, threats, and risks in the software application and ensures that data and resources are protected from potential breaches.

Example: Healthcare Management System

For a healthcare management system, security testing would involve:

- **Authentication and Authorization**: Ensuring that only authorized users can access sensitive patient data and perform specific actions based on their roles.
- **Data Encryption**: Verifying that patient data is encrypted both in transit and at rest to prevent unauthorized access.
- **Vulnerability Scanning**: Identifying and addressing potential vulnerabilities, such as SQL injection, cross-site scripting (XSS), and other common security issues.

Test Scenario: Conducting penetration testing to simulate potential cyber-attacks and assess the system's ability to resist unauthorized access and data breaches.

Summary

- **Functional Testing**: Verifies that the software performs its intended functions correctly (e.g., online banking system's transaction features).
- **Performance Testing**: Assesses how the software performs under various conditions, including high load and stress (e.g., e-commerce website during high-traffic events).
- **Security Testing**: Identifies vulnerabilities and ensures data protection against potential threats (e.g., healthcare management system's data security measures).

22.5 TEST STRATEGIES FOR WEBAPPS