CHAPTER 17

SOFTWARE TESTING

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

WHAT TESTING SHOWS

STRATEGIC APPROACH

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

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. Rosem (Rosem) states this another way:
  - Verification: "Are we building the product right?"
  - Validation: "Are we building the right product?"

WHO TESTS THE SOFTWARE?

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

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

TESTING STRATEGY

- We begin by "testing-in-the-small" and move toward "testing-in-the-large."
- For conventional software:
  - The module (component) is our initial focus
  - Integration of modules follows
- For CO software:
  - Our focus when "testing-in-the-small" changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

STRATEGIC ISSUES

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes "rapid cycle testing."
- Build "robust" software that is designed to test itself.
- Use effective technical reviews as a filter prior to testing.
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.
UNIT TESTING

module to be tested

interface
local data structures
boundary conditions
independent paths
test cases
error handling paths

RESULTS

driver

local data structures
boundary conditions
independent paths
test cases
error handling paths

INTEGRATION TESTING
STRATEGIES

Options:
- the "big bang" approach
- an incremental construction strategy

SANDWICH TESTING

Top modules are tested with stubs

Worker modules are grouped into builds and integrated

REGRESSION TESTING

Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side-effects.

- Whenever software is corrected, some aspect of the software configuration of the program, its documentation, or the data that support it is changed.
- Regression testing helps to ensure that changes due to testing or other reasons do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually by re-executing a subset of all test cases or using automated capture/playback tools.

SMOKE TESTING

- A common approach for creating "daily builds" for product software.
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a "build".
  - A build is a collection of all data files, libraries, reusable modules, and other integrated data that are important to software development.
  - A series of tests is designed to expose errors that prevent the build from properly performing its function.
  - The intent should be to uncover "showstopper" errors that have the highest likelihood of derailing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
  - The integration approach may be top-down or bottom-up.
OBJECT-ORIENTED TESTING

- begins by evaluating the correctness and consistency of the analysis and design models
- testing strategy changes
  - the concept of the 'unit' broadens due to encapsulation
  - integration focuses on classes and their execution across a 'thread' or in the context of a usage scenario
- validation uses conventional black box methods
- test case design draws on conventional methods, but also encompasses special features

TESTING THE CRC MODEL

1. Revisit the CRC model and the object-relationship model.
2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
3. Invent the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
4. Using the invented connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
5. Determine whether widely requested responsibilities might be combined into a single responsibility.
6. Steps 1 to 5 are applied iteratively to each class and through each evolution of the analysis model.

OO TESTING STRATEGY

- class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- integration applied three different strategies
  - thread-based testing integrates the set of classes required to respond to one input event
  - use-based testing integrates the set of classes required to respond to one use case
  - cluster testing integrates the set of classes required to demonstrate one collaboration

HIGH ORDER TESTING

- Validation testing
  - focus is on software requirements
- System testing
  - focus is on system integration
- Acceptance testing
  - focus is on customer usage
- Recovery testing
  - focus is on recovery testing
- Security testing
  - verify that protection mechanisms built into a system will in fact protect it from improper penetration
- Stress testing
  - stress a system in a manner that demands resources in abnormality, intensity, frequency, or volume
- Performance testing
  - test the run-time performance of software within the context of an integrated system

DEBUGGING: A DIAGNOSTIC PROCESS

THE DEBUGGING PROCESS

DEBUGGING EFFORT

SYMPTOMS & CAUSES

- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent

DEBUGGING TECHNIQUES

- brute force / testing
- backtracking
- induction
- deduction
CORRECTING THE ERROR

- Is the cause of the bug manifested in another part of the program? In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
- What “next bug” might be introduced by the fix I’m about to make? Before the correction is made, the source code (or better, the design) should be evaluated to assess coupling of logic and data structures.
- What could we have done to prevent this bug in the first place? This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

TESTABILITY

- Operability—it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

TEST CASE DESIGN

“Bugs lurk in corners and congregate at boundaries...”

Boris Beizer

OBJECTIVE to uncover errors
CRITERIA in a complete manner
CONSTRAINT with a minimum of effort and time

WHAT IS A “GOOD” TEST?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”.
- A good test should be neither too simple nor too complex

EXHAUSTIVE TESTING

There are $10^9$ possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!

SELECTIVE TESTING

INTERNAL AND EXTERNAL VIEWS

- Any engineered product (and most other things) can be tested in one of two ways:
  - Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
  - Knowing the internal workings of a product, tests can be conducted to ensure that “all gears mesh,” that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

FINAL THOUGHTS

- Think—before you act to correct
- Use tools to gain additional insight
- If you’re at an impasse, get help from someone else
- Once you correct the bug, use regression testing to uncover any side effects

CHAPTER 18

Testing Conventional Applications
Software Engineering: A Practitioner’s Approach, 3rd
by Roger S. Pressman
SOFTWARE TESTING

WHITE-BOX TESTING

WHY COVER?
- Logic errors and incorrect assumptions are inversely proportional to a path's execution probability.
- We often believe that a path is not likely to be executed; in fact, reality is often counterintuitive.
- Typographical errors are random; it's likely that untested paths will contain some.

BASIS PATH TESTING

First, we compute the cyclomatic complexity:
- Number of simple decisions + 1
- Number of enclosed areas + 1
In this case, \( V(G) = 4 \)

CYCLOMATIC COMPLEXITY

A number of industry studies have indicated that the higher \( V(G) \), the higher the probability of errors.

BASIS PATH TESTING NOTES

- You don't need a flow chart, but the picture will help when you trace program paths.
- Count each simple logical test, compound tests count as 2 or more.
- Basis path testing should be applied to critical modules.

DERIVING TEST CASES

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

GRAPH MATRICES

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating programs control structure during testing.
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.

DATA FLOW TESTING

- The data flow testing method [Fisk03] selects test paths of a program according to the locations of definitions and uses of variables in the program.
- Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number:
  - DF(S) = {X | Statement S contains a definition of X}
  - USE(S) = {X | Statement S contains a use of X}

A definition-use (DU) chain of variable X is of the form (X, Y, Z), where X, Y, and Z are statement numbers, X is in DF(Y), and the definition of X in statement S is live at statement Z.

LOOP TESTING

- Loop testing: simple loops

  Minimum conditions — 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 m < n
  5. (n-1), n, and (n+1) passes through the loop

  where n is the maximum number of allowable passes

LOOP TESTING: NESTED LOOPS

- Nested Loops

  Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.
  - Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.
  - Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

  Concatenated Loops

  If the loops are independent of one another then treat each as a simple loop else treat as nested loops

  for example, the final loop counter value of loop 1 is used to initialize loop 2.

BLACK-BOX TESTING

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will initiate good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries 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" to be the broadest possible context. It includes both atomic objects (e.g., numbers, strings) and objects (e.g., variable names) and object-oriented elements of computer software.

EQUIVALENCE PARTITIONING

- User actions
- Failure modes
- Input conditions
- Test cases
- Test input data
SAMPLE EQUIVALENCE CLASSES

Valid data
- user supplied commands
- responses to system prompts
- file names
- computational data
- physical parameters
- boundary values
- initialization values
- output data formatting
- responses to error messages
- graphical data (e.g., mouse picks)

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

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

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.

SOFTWARE TESTING PATTERNS

- Testing patterns are described in much the same way as design patterns (Chapter 12).
- 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. [Kan01]

CHAPTER 19

- Testing Object-Oriented Applications

Software Engineering: A Practitioner’s Approach, 7th
by Roger S. Pressman

OO TESTING

- To adequately test OO systems, three things must be done:
  - the definition of testing must be broadened to include error discovery techniques applied to object-oriented analysis and design models
  - 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,

COMPARISON TESTING

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
- Separate software engineering teams develop independent versions of an application using the same specification
- Each version can be tested with the same test data to ensure that all provide identical output
- Then all versions are executed in parallel with real-time comparison of results to ensure consistency
CORRECTNESS OF OO MODELS

- During analysis and design, semantic correctness can be assessed based on the model's conformance to the real world problem domain.
- If the model accurately reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed) then it is semantically correct.
- To determine whether the model does, in fact, reflect real world requirements, it should be presented to problem domain experts who will examine the class definitions and hierarchy for omissions and ambiguity.
- Class relationships (instance connections) are evaluated to determine whether they accurately reflect real-world object connections.

CLASS MODEL CONSISTENCY

- Revise the CRC model and the object-relationship model.
- Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
- Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
- Using the inverted connections examined in the preceding step, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
- Determine whether widely requested responsibilities might be combined into a single responsibility.

OO TESTING STRATEGIES

- Unit testing
  - the concept of the unit changes
  - the smallest testable unit is the encapsulated class
  - simple operations can no longer be tested in isolation (the conventional view of unit testing) but rather, as part of a class
- Integration testing
  - Test-based testing integrates the set of classes required to respond to one input or event for the system.
  - Goal-based testing begins the construction of the system by testing those classes that depend upon others.
  - Test-based testing tests individual classes, while the goal-based testing tests the complete set of classes. After the independent classes are tested, the next level of classes, called dependent classes
- Cluster testing (BTC) defines a cluster of collaborating classes determined by examining the CRC and object-relationships models (e.g., by designing test cases that attempt to uncover errors in the collaborations.

OO TESTING STRATEGIES

- Validation testing
  - details of class connections disappear
  - draw upon use cases (Chapters 5 and 6) that are part of the requirements model
- Conventional black-box testing methods (Chapter 18) can be used to drive validation tests

OOP METHODS

Barard (Ber93) proposes the following approach:
1. Each test case should be uniquely identified and should be explicitly associated with the class being tested.
2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain business-level test steps.
   - a list of steps that will be performed on the class (this is not a list of methods that will be executed)
   - a list of steps that will be performed on the environment that the class will run in
   - a list of steps that will be performed on the tester (i.e., change in the tester's environment to run the class)
   - a list of steps that will be performed on the test tool (i.e., change in the test tool environment to run the class)
   - a list of steps that will be performed on the system (i.e., change in the system environment to run the class)
   - a list of steps that will be performed on the tester (i.e., change in the tester's environment to run the test)

TESTING METHODS

- Fault-based testing
  - The tester looks for observable 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 override 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, rather than what the product does. The goal is to test the tasks the user is supposed to do before the user has to perform them, making them easier to test.

OOP METHODS: RANDOM TESTING

- Random testing
  - Identify operations applicable to a class
  - Define constraints on their use
  - Identify a minimum test sequence
    - an operation sequence that defines the minimum life history of the class (object)
    - generates a variety of random (but valid) test sequences
    - exercise other (more complex) class instance life histories

OOP METHODS: PARTITION TESTING

- 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
  - category and test operators based on their ability to change the state of a class
  - attribute-based partitioning
  - attributes and test operators based on the attributes that they use
  - category-based partitioning
  - categories and test operators based on the generic function each performs

OOP METHODS: INTER-CLASS TESTING

- Inter-class testing
  - For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes.
  - for each message that is generated, determine the class that it invokes, the corresponding operator in the server object, and for each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
  - for each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.
OOT METHODS: BEHAVIOR TESTING

The tests to be designed should achieve all state coverage [KIR94]. That is, the operation sequence should cause the Account class to make transition through all allowable states.

TESTING QUALITY DIMENSIONS-I

- Content is evaluated at both a syntactic and semantic level:
  - Syntactic level—spelling, punctuation and grammar are assessed for text-based documents.
  - Semantic level—correctness of information presented, consistency (across the entire content object and within objects), and bias of subjective influence are assessed.

- Functionality is tested for completeness, flexibility, and general conformance to appropriate implementation standards (e.g., Java or .NET language standards).

- Structure is assessed to ensure that it:
  - properly delivers WebApp content and function
  - is maintainable
  - can be supported as new content or functionality is added.

CHAPTER 20

- Testing Web Applications
  - Software Engineering: A Practitioner’s Approach, 7/e
  - by Roger S. Pressman

TESTING QUALITY DIMENSIONS-II

- Usability is tested to ensure that each category of user:
  - is supported by the interface
  - can learn and apply all required navigation syntax and semantics

- Navigability is tested to ensure that:
  - all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, erroneous links).

- Performance is tested under a variety of operating conditions, configurations, and load to ensure that:
  - the system is responsive to user interaction
  - the system handles extreme loads without unacceptable operational degradation.

WEBAPP TESTING STRATEGY-I

- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and navigation mechanics.
- Selected functional components are unit tested.

TESTING QUALITY DIMENSIONS-III

- Compatibility is tested by executing the WebApp in a variety of different host configurations on both the client and server sides.
  - The intent is to find errors that are specific to a unique host configuration.

- Interoperability is tested to ensure that the WebApp properly interfaces with other applications and/or databases.

- Security is tested by assessing potential vulnerabilities and attempting to exploit each.
  - Any successful penetration attempt is deemed a security failure.

WEBAPP TESTING STRATEGY-II

- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.

- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.

- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end-users.
  - The results of their interaction with the system are evaluated for context and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

TESTING QUALITY DIMENSIONS-IV

- Functionality is tested for completeness, flexibility, and general conformance to appropriate implementation standards (e.g., Java or .NET language standards).

- Structure is assessed to ensure that it:
  - properly delivers WebApp content and function
  - is maintainable
  - can be supported as new content or functionality is added.

- Usability is tested to ensure that each category of user:
  - is supported by the interface
  - can learn and apply all required navigation syntax and semantics

- Navigability is tested to ensure that:
  - all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, erroneous links).

- Performance is tested under a variety of operating conditions, configurations, and load to ensure that:
  - the system is responsive to user interaction
  - the system handles extreme loads without unacceptable operational degradation.

THE TESTING PROCESS

- The testing process involves several key steps:
  - Definition: Identifying the goals and requirements for testing.
  - Planning: Developing a comprehensive testing strategy.
  - Execution: Conducting the tests and analyzing the results.
  - Reporting: Communicating the findings to stakeholders.
  - Improvement: Continuously refining the testing process based on feedback and analysis.

- Testing is an iterative process that involves multiple rounds of review, testing, and refinement.

- Effective testing involves not only identifying and fixing defects but also ensuring that the system meets its intended specifications and requirements.

- Testing strategies should be tailored to the specific needs and constraints of the project, taking into account factors such as the size of the system, the available resources, and the nature of the risks involved.

- Continuous testing is crucial for maintaining the reliability and performance of software systems, as it helps to identify and address issues early in the development cycle, reducing the costs and delays associated with bug fixes later on.

- Testing should be an integral part of the software development process, with rigorous testing practices in place at every stage to ensure the quality and robustness of the final product.

- Effective testing requires a combination of technical skills, domain knowledge, and excellent communication abilities to effectively collaborate with developers, stakeholders, and other team members.

- The success of testing efforts depends on the ability to continuously improve and adapt testing strategies based on feedback and real-world experiences.
CONTENT TESTING

- Content testing has three important objectives:
  - to uncover syntactic errors (e.g., typos, grammar mistakes) in text-based documents, graphical representations, and other media
  - to uncover semantic errors (e.g., errors in the accuracy or completeness of information) in any content object presented as navigation options, and
  - to find errors in the organization or structure of content that is presented to the end-user.

USER INTERFACE TESTING

- Interface features are tested to ensure that design rules, aesthetics, and related visual content is available for the user without error.
- Individual interface mechanisms are tested in a manner that is analogous to unit testing.
- Each interface mechanism is tested within the context of a use case or scenario for a specific user category.
- The complete interface is tested against selected use cases and scenarios to uncover errors in the semantics of the interface.
- The interface is tested within a variety of environments (e.g., browsers) to ensure that it will be compatible.

TESTING INTERFACE MECHANISMS-I

- Link—navigation mechanisms that link the user to some other content object or function.
- Forms—structured document containing input fields that are filled in by the user. The data contained in the fields are used as input to one or more WebApp functions.
- Client-side scripting—a set of programming commands in a scripting language (e.g., JavaScript) that handle information input via forms or other user interactions.
- Dynamic HTML—leads to content objects that are manipulated on the client side using scripting or cascading style sheets (CSS).
- Client-side pop-up windows—small windows that pop-up without user interaction. These windows can be content-oriented and may require some form of user interaction.

USABILITY TESTS

- Design by Web team ..., executed by end-users
- Testing sequence...
  - Define a set of usability testing categories and identify goals for each
  - Design tests that will enable each goal to be evaluated
  - Select participants who will conduct the tests
  - Instrument the WebApp with an interface testing tool
  - Develop a mechanism for assessing the usability of the WebApp
  - Different levels of abstraction:
    - the usability of a specific interface mechanism (e.g., a form) can be assessed
    - the usability of a complete Web page (comprising interface mechanisms) can be evaluated
    - the usability of the complete WebApp can be considered.

COMPATIBILITY TESTING

- Compatibility testing is to define a set of "commonly encountered" client side computing configurations and their variants
- Create a test structure identifying:
  - each computing platform
  - typical devices
  - the operating system and software
  - the browser and add-ons
  - link Internet connection speeds
  - similiar information
- Define a series of compatibility validation tests:
  - derived from existing interface tests, navigation tests, performance tests, and security tests
  - intent of these tests is to uncover errors or execution problems that can be traced to configuration differences

DATABASE TESTING

- Tests are defined for each layer
- Details of data sent by the server and stored by the browser as a consequence of a specific user interaction
- The content of the data is window-specific (e.g., user identification data or a list of items that have been selected for purchase by the user)
- Application specific interface mechanisms include one or more "map" interface mechanisms such as a shopping cart, credit card processing, or a shipping cost calculator.

COMPONENT-LEVEL TESTING

- Focuses on a set of tests that attempt to uncover errors in WebApp functions
- Conventional black-box and white-box test case design methods can be used
- Database testing is often an integral part of the component-testing regime
NAVIGATION TESTING

- The following navigation mechanisms should be tested:
  - Navigation links; these mechanisms include internal links
    within the WebApp, external links to other WebApps, and
    anchors within a specific Web page.
  - Redirections—these links are often set when a user requests
    a non-existent URL or when a link that a whole destination
    has been removed or whose name has changed.
  - Browser—although bookmarks are a browser function,
    the WebApp should be tested to ensure that a meaningful
    page title can be extracted as the bookmark is created.
  - Frames and Formas—tested for correct content, proper
    field size, labeling, download performance, and browser
    support.
  - Site map(s)—each site map entry should be tested to
    ensure that the link takes the user to the proper content
    or functionality.
  - Internal search engine: search engine indexing validates
    the accuracy and completeness of the search, the error-
    handling properties of the search engine, and advanced
    search features.

CONFIGURATION TESTING

- Server-side
  - Is the WebApp fully compatible with the server OS?
  - Are system files, directories, and config data stored correctly
    where they are operational?
  - Does the security measure (e.g., firewalls or encryption) allow
    the WebApp to execute and services without interference or
    performance degradation?
  - Does the WebApp run when the distributed server configuration
    is up and running that has been deployed?
  - Is the WebApp properly integrated with databases software?
  - Is the WebApp running with different versions of database software?
  - Does the WebApp scripts execute properly?
  - Have system administrators been examined for their affect on
    WebApp operations?
  - If proxy servers are used, have differences in configuration been
    addressed with on-line testing?

- Client-side
  - Hardware—CPU, memory, storage and printing devices
  - Operating systems—Linux, Macintosh OS, Microsoft Windows, a
    multi-OS environment.
  - Browsers—Internet Explorer, Mozilla/Netscape, Opera, Safari, and
    others.
  - User interface components—Active X, Java applets and others
    Plug-ins—Quicktime, RealPlayer, and many others.
  - Connectivity—cable, DSL, regular modem, T1.
  - The number of configuration variables must be reduced to a
    manageable number.

SECURITY TESTING

- Designed to probe vulnerabilities of the client-side
  environment, the network communications that occur as data are passed from
  client to server and back again, and the server-side environment.
- On the client-side, vulnerabilities can often be traced
  to pre-existing bugs in browsers, e-mail programs, or
  communication software.
- On the server-side, vulnerabilities include denial-of-
  service attacks and malicious scripts that can be
  passed along to the client-side or used to disable server
  operations.

PERFORMANCE TESTING

- Does the server response time degrade to a point where it is
  noticeable and unacceptable?
- At what point (in terms of users, transactions or data loading) does
  performance become unacceptable?
- What system components are responsible for performance
  degradation?
- What is the average response time for users under a variety of
  loading conditions?
- Does performance degradation have any impact on system security?
- Is WebApp reliability or accuracy affected as the load on the
  system grows?
- What happens when loads that are greater than maximum server
  capacity are approached?

LOAD TESTING

- The intent is to determine how the WebApp and its
  server-side environment will respond to various loading
  conditions.
- N, the number of concurrent users
- T, the number of online transactions per unit of time
- D, the data load processed by the server per transaction
- Overall throughput, P, is computed in the following manner:
  \[ P = N \times T \times D \]