Chapters 18 - 20

Highlights & Summary

Testing , Reliability, & Quality

- Testing cannot prove a product has no errors
  - It can only find errors and/or show they have been fixed
  - … Or fail to find errors
  - It cannot prove a product is error free
- Reliability
  - probability of failure free operation of a computer program in a specified environment for a specified time period
  - Software reliability problems can usually be traced back to errors in design or implementation
- Statistical Quality Assurance
  - Collect, categorize and trace each error
  - Isolate the main causes
    - 80% errors from 20% source
    - Also called “viral few causes”
  - Focus on fixing those error sources
  - Result: 80% reduction in error

Software Testing Objectives

- Testing is the process of executing a program with the **intent** of finding errors.
- A good test case is one with a high probability of finding an as-yet undiscovered error.
- A successful test is one that discovers an as-yet-undiscovered error.

White Box vs Black Box Testing

- White Box
  - Knowledge of inner workings of the program
  - Unit Testing is an example
  - EVERY statement/condition tested at least once
  - ➢ Error likelihood is inversely proportional to path's execution probability
- Black Box
  - No implicit knowledge of inner workings of the program
  - Independent testers run "black box" tests (they don't know the code)
- Modern testing has dropped these terms
  - Functional: Does it do what it says it will do?
  - Operational: Does it behave as expected?
  - Performance: Does it scale?
  - General user interface and system response
  - Boundary Value Analysis (if normal limits = a, b then test a±1 and b±1)

Good Test Attributes

- 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 not be too simple or too complex.

Object Oriented Software Test Strategies

- 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.

Testing Object-Based Software

- *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* [McG94]
  - 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.

Object-Oriented Testing Approach

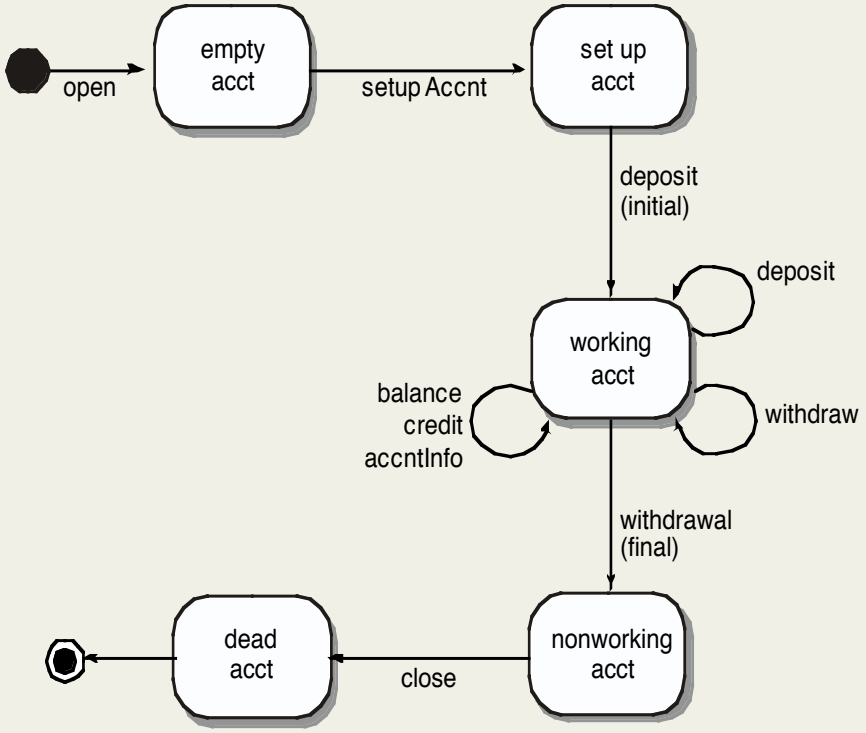
- Each test case should be uniquely identified and should be explicitly associated with the class to be tested,
- The purpose of the test should be stated,
- A list of testing steps should be developed for each test and should contain [BER93]:
  - a list of specified states for the object that is to be tested
  - a list of messages and operations that will be exercised as a consequence of the test
  - a list of exceptions that may occur as the object is tested
  - a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
  - supplementary information that will aid in understanding or implementing the test.

Object-Oriented Test Methods

- 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: Behavior Testing

- **The tests to be designed should achieve coverage of all possible states**
- **Ex: the operation sequences should cause the Account class to transition through all allowable states**



Web App Errors

- Because many types of WebApp tests uncover problems that are first evidenced on the client side, you often see a symptom of the error, not the error itself.
- Because a WebApp is implemented in a number of different configurations and within different environments, it may be difficult or impossible to reproduce an error outside the environment in which the error was originally encountered.
- Although some errors are the result of incorrect design or improper HTML (or other programming language) coding, many errors can be traced to the WebApp configuration.
- Because WebApps reside within a client/server architecture, errors can be difficult to trace across three architectural layers: the client, the server, or the network itself.
- Some errors are due to the *static operating environment* (i.e., the specific configuration in which testing is conducted), while others are attributable to the dynamic operating environment (i.e., instantaneous resource loading or time-related errors).

Web App Testing -1

- *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 related objects) and lack of ambiguity are all assessed.
- *Function* is tested for correctness, instability, and general conformance to appropriate implementation standards (e.g.,Java or XML language standards).
- *Structure* is assessed to ensure that it
  - properly delivers WebApp content and function
  - is extensible
  - can be supported as new content or functionality is added.

Web App Testing - 2

- *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 loading to ensure that
  - the system is responsive to user interaction
  - the system handles extreme loading without unacceptable operational degradation

Web App Testing - 3

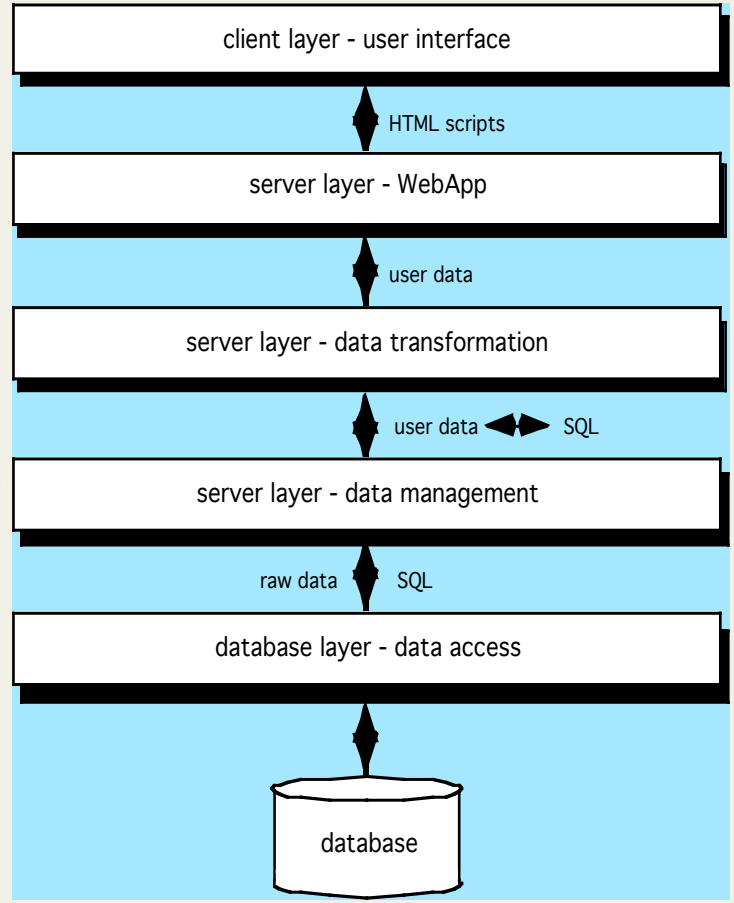
- *Compatibility*
  - tested by executing the WebApp in a variety of different host configurations on both the client and server sides.
  - The intent is to find errors that are specific to a unique host configuration.
- *Interoperability*
  - tested to ensure that the WebApp properly interfaces with other applications and/or databases.
- *Security*
  - tested by assessing potential vulnerabilities and attempting to exploit each.
  - Any successful penetration attempt is deemed a security failure.

Assessing Content Semantics

- Is the information factually accurate?
- Is the information concise and to the point?
- Is the layout of the content object easy for the user to understand?
- Can information embedded within a content object be found easily?
- Have proper references been provided for all information derived from other sources?
- Is the information presented consistent internally and consistent with information presented in other content objects?
- Is the content offensive, misleading, or does it open the door to litigation?
- Does the content infringe on existing copyrights or trademarks?
- Does the content contain internal links that supplement existing content? Are the links correct?
- Does the aesthetic style of the content conflict with the aesthetic style of the interface?

Database Testing

Tests are defined for each layer



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 NSU for a specific user category.
- The complete interface is tested against selected use-cases and NSUs 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

- *Links*—navigation mechanisms that link the user to some other content object or function.
- *Forms*—a structured document containing blank 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 list of programmed 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.

Testing Interface Mechanisms-II

- *CGI scripts*—a common gateway interface (CGI) script implements a standard method that allows a Web server to interact dynamically with users (e.g., a WebApp that contains forms may use a CGI script to process the data contained in the form once it is submitted by the user).
- *Streaming content*—rather than waiting for a request from the client-side, content objects are downloaded automatically from the server side. This approach is sometimes called "push" technology because the server pushes data to the client.
- *Cookies*—a block of data sent by the server and stored by a browser as a consequence of a specific user interaction. The content of the data is WebApp-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 "macro" interface mechanisms such as a shopping cart, credit card processing, or a shipping cost calculator.

Usability Tests

- Design by WebE 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 participants' interaction with the WebApp while testing is conducted.
  - 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 (encompassing interface mechanisms, data objects and related functions) 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 tree structure identifying
  - each computing platform
  - typical display devices
  - the operating systems supported on the platform
  - the browsers available
  - likely Internet connection speeds
  - similar information.
- Derive 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.

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.
  - *Redirects*—these links come into play when a user requests a non-existent URL or selects a link whose destination has been removed or whose name has changed.
  - *Bookmarks*—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 framesets*—tested for correct content, proper layout and sizing, download performance, and browser compatibility
  - *Site maps*—Each site map entry should be tested to ensure that the link takes the user to the proper content or functionality.
  - *Internal search engines*—Search engine testing validates the accuracy and completeness of the search, the error-handling properties of the search engine, and advanced search features

Testing Navigation Semantics-I

- Is the NSU achieved in its entirety without error?
- Is every navigation node (defined for a NSU) reachable within the context of the navigation paths defined for the NSU?
- If the NSU can be achieved using more than one navigation path, has every relevant path been tested?
- If guidance is provided by the user interface to assist in navigation, are directions correct and understandable as navigation proceeds?
- Is there a mechanism (other than the browser 'back' arrow) for returning to the preceding navigation node and to the beginning of the navigation path.
- Do mechanisms for navigation within a large navigation node (i.e., a long web page) work properly?
- If a function is to be executed at a node and the user chooses not to provide input, can the remainder of the NSU be completed?

Testing Navigation Semantics-II

- If a function is executed at a node and an error in function processing occurs, can the NSU be completed?
- Is there a way to discontinue the navigation before all nodes have been reached, but then return to where the navigation was discontinued and proceed from there?
- Is every node reachable from the site map? Are node names meaningful to end-users?
- If a node within an NSU is reached from some external source, is it possible to process to the next node on the navigation path. Is it possible to return to the previous node on the navigation path?
- Does the user understand his location within the content architecture as the NSU is executed?

Configuration Testing

- Server-side
  - Is the WebApp fully compatible with the server OS?
  - Are system files, directories, and related system data created correctly when the WebApp is operational?
  - Do system security measures (e.g., firewalls or encryption) allow the WebApp to execute and service users without interference or performance degradation?
  - Has the WebApp been tested with the distributed server configuration (if one exists) that has been chosen?
  - Is the WebApp properly integrated with database software? Is the WebApp sensitive to different versions of database software?
  - Do server-side WebApp scripts execute properly?
  - Have system administrator errors been examined for their affect on WebApp operations?
  - If proxy servers are used, have differences in their configuration been addressed with on-site testing?

Configuration Testing

- Client-side
  - *Hardware*—CPU, memory, storage and printing devices
  - *Operating systems*—Linux, Macintosh OS, Microsoft Windows, a mobile-based OS
  - *Browser software*—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 an 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 applied?

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 on-line transactions per unit of time
  - *D,* the data load processed by the server per transaction
- Overall throughput, *P,* is computed in the following manner:
    - $P = N \times T \times D$

Stress Testing

- Does the system degrade 'gently' or does the server shut down as capacity is exceeded?
- Does server software generate "server not available" messages? More generally, are users aware that they cannot reach the server?
- Does the server queue requests for resources and empty the queue once capacity demands diminish?
- Are transactions lost as capacity is exceeded?
- Is data integrity affected as capacity is exceeded?
- What values of *N, T,* and *D* force the server environment to fail? How does failure manifest itself? Are automated notifications sent to technical support staff at the server site?
- If the system does fail, how long will it take to come back on-line?
- Are certain WebApp functions (e.g., compute intensive functionality, data streaming capabilities) discontinued as capacity reaches the 80 or 90 percent level?