CHAPTER 13 SOFTWARE TESTING STRATEGIES

Overview

This chapter describes several approaches to testing software. Software testing must be planned carefully to avoid wasting development time and resources. Testing begins "in the small" by focusing on software components and progresses "to the large" by considering system functionality as a whole. Initially individual components are tested and debugged. After the individual components have been tested and added to the system, integration testing takes place. Once the full software product is completed, system testing is performed. The Test Specification document should be reviewed like all other software engineering work products. A sample Test Specification document appears on the SEPA Web site. The details of testing techniques and test case construction are described in the next chapter of the text.

Strategic Approach to Software Testing

- Testing begins at the component level and works outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- The developer of the software conducts testing and may be assisted by independent test groups for large projects.
- Testing and debugging are different activities.
- Debugging must be accommodated in any testing strategy.

Verification and Validation

- Make a distinction between verification (are we building the product right?) and validation (are we building the right product?)
- Software testing is only one element of Software Quality Assurance (SQA)
- Quality must be built in to the development process, you can't use testing to add quality after the fact

Organizing for Software Testing

- The role of the Independent Test Group (ITG) is to remove the conflict of interest inherent when the builder is testing his or her own product.
- Misconceptions regarding the use of independent testing teams are
  - The developer should do no testing at all
  - Software is tossed "over the wall" to people to test it mercilessly
  - Testers are not involved with the project until it is time for it to be tested
- The developer and ITGC must work together throughout the software project to ensure that thorough tests will be conducted

Software Testing Strategy for Traditional Software Architectures
• Unit Testing - makes heavy use of testing techniques that exercise specific control paths to detect errors in each software component individually
• Integration Testing - focuses on issues associated with verification and program construction as components begin interacting with one another
• Validation Testing - provides assurance that the software validation criteria (established during requirements analysis) meets all functional, behavioral, and performance requirements
• System Testing - verifies that all system elements mesh properly and that overall system function and performance has been achieved

Software Testing Strategy for Object-Oriented Architectures

• Unit Testing - components being tested are classes not modules
• Integration Testing - as classes are integrated into the architecture regression tests are run to uncover communication and collaboration errors between objects
• Systems Testing - the system as a whole is tested to uncover requirement errors

Strategic Testing Issues

• Specify product requirements in a quantifiable manner before testing starts.
• Specify testing objectives explicitly.
• Identify categories of users for the software and develop a profile for each.
• Develop a test plan that emphasizes rapid cycle testing.
• Build robust software that is designed to test itself.
• Use effective formal reviews as a filter prior to testing.
• Conduct formal technical reviews to assess the test strategy and test cases.
• Develop a continuous improvement approach for the testing process.

Unit Testing

• Module interfaces are tested for proper information flow.
• Local data are examined to ensure that integrity is maintained.
• Boundary conditions are tested.
• Basis (independent) path are tested.
• All error handling paths should be tested.
• Drivers and/or stubs need to be developed to test incomplete software.

Integration Testing

• Top-down integration testing
  1. Main control module used as a test driver and stubs are substitutes for components directly subordinate to it.
  2. Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
  3. Tests are conducted as each component is integrated.
4. On completion of each set of tests and other stub is replaced with a real component.
5. Regression testing may be used to ensure that new errors not introduced.
   - Bottom-up integration testing
     1. Low level components are combined into clusters that perform a specific software function.
     2. A driver (control program) is written to coordinate test case input and output.
     3. The cluster is tested.
     4. Drivers are removed and clusters are combined moving upward in the program structure.
   - Regression testing - used to check for defects propagated to other modules by changes made to existing program
     1. Representative sample of existing test cases is used to exercise all software functions.
     2. Additional test cases focusing software functions likely to be affected by the change.
     3. Tests cases that focus on the changed software components.
   - Smoke testing
     1. Software components already translated into code are integrated into a build.
     2. A series of tests designed to expose errors that will keep the build from performing its functions are created.
     3. The build is integrated with the other builds and the entire product is smoke tested daily (either top-down or bottom integration may be used).

General Software Test Criteria

- Interface integrity - internal and external module interfaces are tested as each module or cluster is added to the software
- Functional validity - test to uncover functional defects in the software
- Information content - test for errors in local or global data structures
- Performance - verify specified performance bounds are tested

Object-Oriented Unit Testing

- smallest testable unit is the encapsulated class or object
- similar to system testing of conventional software
- do not test operations in isolation from one another
- driven by class operations and state behavior, not algorithmic detail and data flow across module interface

Object-Oriented Integration Testing

- focuses on groups of classes that collaborate or communicate in some manner
- integration of operations one at a time into classes is often meaningless
- thread-based testing - testing all classes required to respond to one system input or event
- use-based testing - begins by testing independent classes (classes that use very few server classes) first and the dependent classes that make use of them
- cluster testing - groups of collaborating classes are tested for interaction errors
- regression testing is important as each thread, cluster, or subsystem is added to the system

Validation Testing

- Pretty much the same for both conventional and object-oriented software
- Focuses on visible user actions and user recognizable outputs from the system
- Validation tests are based on the use-case scenarios, the behavior model, and the event flow diagram created in the analysis model
  - Must ensure that each function or performance characteristic conforms to its specification.
  - Deviations (deficiencies) must be negotiated with the customer to establish a means for resolving the errors.
- Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.

Acceptance Testing

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha test - version of the complete software is tested by customer under the supervision of the developer at the developer's site
- Beta test - version of the complete software is tested by customer at his or her own site without the developer being present

System Testing

- Recovery testing - checks the system's ability to recover from failures
- Security testing - verifies that system protection mechanism prevent improper penetration or data alteration
- Stress testing - program is checked to see how well it deals with abnormal resource demands (i.e., quantity, frequency, or volume)
- Performance testing - designed to test the run-time performance of software, especially real-time software

Debugging

- Debugging (removal of a defect) occurs as a consequence of successful testing.
- Some people are better at debugging than others.
- Common approaches:
- **Brute force** - memory dumps and run-time traces are examined for clues to error causes
- **Backtracking** - source code is examined by looking backwards from symptom to potential causes of errors
- **Cause elimination** - uses binary partitioning to reduce the number of locations potential (where errors can exist)

**Bug Removal Considerations**

- Is the cause of the bug reproduced in another part of the program?
- What “next bug” might be introduced by the fix that is being proposed?
- What could have been done to prevent this bug in the first place?