Software Testing Techniques: Overview

The importance of software testing to software quality can not be overemphasized. Once source code has been generated, software must be tested to allow errors to be identified and removed before delivery to the customer. While it is not possible to remove every error in a large software package, the software engineer's goal is to remove as many as possible early in the software development cycle. It is important to remember that testing can only find errors, it cannot prove that a program is bug free. Two basic test techniques involve testing module input/output (black-box) and exercising internal logic of software components (white-box). Formal technical reviews by themselves can not find allow software defects, test data must also be used. For large software projects, separate test teams may be used to develop and execute the set of test cases used in testing. Testing must be planned and designed. The SEPA web site contains the template for a generic test plan.

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.

Software Testing Principles

- All tests should be traceable to customer requirements.
- Tests should be planned long before testing begins.
- The Pareto principle (80% of all errors will likely be found in 20% of the code) applies to software testing.
- Testing should begin in the small and progress to the large.
- Exhaustive testing is not possible.
- To be most effective, testing should be conducted by an independent third party.

Software Testability Checklist

- Operability (the better it works the more efficiently it can be tested)
- Observabilty (what you see is what you test)
- Controllability (the better software can be controlled the more testing can be automated and optimized)
- Decomposability (by controlling the scope of testing, the more quickly problems can be isolated and retested intelligently)
- Simplicity (the less there is to test, the more quickly we can test)
- Stability (the fewer the changes, the fewer the disruptions to testing)
- Understandability (the more information known, the smarter the testing)

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.

Test Case Design Strategies

- Black-box or behavioral testing (knowing the specified function a product is to perform and demonstrating correct operation based solely on its specification without regard for its internal logic)
- White-box or glass-box testing (knowing the internal workings of a product, tests are performed to check the workings of all insdependent logic paths)

Basis Path Testing

- White-box technique usually based on the program flow graph
- The cyclomatic complexity of the program computed from its flow graph using the formula V(G) = E – N + 2 or by counting the conditional statements in the PDL representation and adding 1
- Determine the basis set of linearly independent paths (the cardinality of this set id the program cyclomatic complexity)
- Prepare test cases that will force the execution of each path in the basis set.

Control Structure Testing

- White-box techniques focusing on control structures present in the software
- Condition testing (e.g. branch testing) focuses on testing each decision statement in a software module, it is important to ensure coverage of all logical combinations of data that may be processed by the module (a truth table may be helpful)
- Data flow testing selects test paths based according to the locations of variable definitions and uses in the program (e.g. definition use chains)
- Loop testing focuses on the validity of the program loop constructs (i.e. simple loops, concatenated loops, nested loops, unstructured loops), involves checking to ensure loops start and stop when they are supposed to (unstructured loops should be redesigned whenever possible)

Graph-based Testing Methods

- Black-box methods based on the nature of the relationships (links) among the program objects (nodes), test cases are designed to traverse the entire graph
- Transaction flow testing (nodes represent steps in some transaction and links represent logical connections between steps that need to be validated)
- Finite state modeling (nodes represent user observable states of the software and links represent transitions between states)
- Data flow modeling (nodes are data objects and links are transformations from one data object to another)
- Timing modeling (nodes are program objects and links are sequential connections between these objects, link weights are required execution times)

Equivalence Partitioning

- Black-box technique that divides the input domain into classes of data from which test cases can be derived
- An ideal test case uncovers a class of errors that might require many arbitrary test cases to be executed before a general error is observed
- Equivalence class guidelines:

  1. If input condition specifies a range, one valid and two invalid equivalence classes are defined
  2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  3. If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined
  4. If an input condition is Boolean, one valid and one invalid equivalence class is defined

Boundary Value Analysis

- Black-box technique that focuses on the boundaries of the input domain rather than its center

- BVA guidelines:

  1. If input condition specifies a range bounded by values a and b, test cases should include a and b, values just above and just below a and b
  2. If an input condition specifies and number of values, test cases should be exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values
  3. Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maxim output reports
  4. If internal program data structures have boundaries (e.g. size limitations), be certain to test the boundaries

Comparison Testing

- Black-box testing for safety critical systems in which independently developed implementations of redundant systems are tested for conformance to specifications
- Often equivalence class partitioning is used to develop a common set of test cases for each implementation

Orthogonal Array Testing

- Black-box technique that enables the design of a reasonably small set of test cases that provide maximum test coverage
- Focus is on categories of faulty logic likely to be present in the software component (without examining the code)
- Priorities for assessing tests using an orthogonal array

  1. Detect and isolate all single mode faults
  2. Detect all double mode faults
  3. Mutimode faults

Specialized Testing

- Graphical user interfaces (see Chapter 31 and the SEPA web checklist)
- Client/server architectures (see Chapter 28)
- Documentation and help facilities (see Chapter 8 and Chapter 15)
- Real-time systems

  1. Task testing (test each time dependent task independently)
  2. Behavioral testing (simulate system response to external events)
  3. Intertask testing (check communications errors among tasks)
  4. System testing (check interaction of integrated system software and hardware)

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" and progresses "to the large". Initially individual components are tested using white box and black box techniques. 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.

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.
- The role of the independent tester is to remove the conflict of interest inherent when the builder is testing his or her own product.
- Testing and debugging are different activities.
- Debugging must be accommodated in any testing strategy.
- Make a distinction between verification (are we building the product right?) and validation (are we building the right product?)

Strategic Testing Issues

- Specify product requirements in a quantifiable manner before testing starts.
- Specify testing objectives explicitly.
- Identify the user classes of 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 (e.g. uses anitbugging).
- Use effective formal reviews as a filter prior to testing.
- Conduct formal technical reviews to assess the test strategy and test cases.

Unit Testing

- Black box and white box testing.
- Module interfaces are tested for proper information flow.
- Local data are examined to ensure that integrity is maintained.
- Boundary conditions are tested.
- Basis path testing should be used.
- 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 in 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 (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)

Validation Testing

- 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 – 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:

1. Brute force (memory dumps and run-time traces are examined for clues to error causes)
2. Backtracking (source code is examined by looking backwards from symptom to potential causes of errors)
3. 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?

REFER TO CHAPTER 17 and 18 *“Pressman. R. S., Software Engineering a practitioners Approach. 5th Edition”* ACCORDINGLY.