CHAPTER 14 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 exist for testing conventional software, testing module input/output (black-box) and exercising the internal logic of software components (white-box). The process of testing object-oriented systems begins with a review of the object-oriented analysis and design models. Once the code is written object-oriented testing begins by testing "in the small" with class testing (class operations and collaborations). As classes are integrated to become subsystems class collaboration problems are investigated. Use-cases from the analysis model are used to uncover software validation errors. For Formal technical reviews by themselves cannot find all 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 Testability Checklist

- Operability - the better it works the more efficiently it can be tested
- Observability - 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 possible logic paths

White-Box Testing Questions

• Can you guarantee that all independent paths within a module will be executed at least once?
• Can you exercise all logical decisions on their true and false branches?
• Will all loops execute at their boundaries and within their operational bounds?
• Can you exercise internal data structures to ensure their validity?

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 program design language (PDL) representation and adding 1
• Determine the basis set of linearly independent paths (the cardinality of this set is the program cyclomatic complexity)
• Prepare test cases that will force the execution of each path in the basis set.

Control Structure Testing

• White-box technique 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)

Black-Box Testing Questions

• How is functional validity tested?
• How is system behavior and performance tested?
• What classes of input will make 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 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 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 maximum 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.

Test Case Design for OO Software

1. Each test case should be uniquely identified and be explicitly associated with a class to be tested.
2. State the purpose of each test.
3. List the testing steps for each test including:
   a. list of states to test for each object involved in the test.
   b. list of messages and operations to be exercised as a consequence of the test.
   c. list of exceptions that may occur as the object is tested.
   d. list of external conditions needed to be changed for the test.
   e. supplementary information required to understand or implement the test.

OO Test Case Design

- White-box testing methods can be applied to testing the code used to implement class operations, but not much else.
- Black-box testing methods are appropriate for testing OO systems.

OO Fault-Based Testing

- Best reserved for operations and the class level.
• Uses the inheritance structure
• Tester examines the OOA model and hypothesizes a set of plausible defects that may be encountered in operation calls and message connections and builds appropriate test cases
• Misses incorrect specification and errors in subsystem interactions

Class Hierarchy and Additional Testing Concerns

• Classes may contain operations that are inherited from super classes
• Subclasses may contain operations that were redefined rather than inherited
• All classes derived from an previously tested base class need to be thoroughly tested

OO Scenario-Based Testing

• Using the user tasks described in the use-cases and building the test cases from the tasks and their variants
• Uncovers errors that occur when any actor interacts with the OO software
• Concentrates on what the use does, not what the product does
• You can get a higher return on your effort by spending more time on reviewing the use-cases as they are created, than spending more time on use-case testing

OO Testing - Surface Structure and Deep Structure

• Testing surface structure - exercising the structure observable by end-user, this often involves observing and interviewing users as they manipulate system objects
• Testing deep structure - exercising internal program structure - the dependencies, behaviors, and communications mechanisms established as part of the system and object design

Class Level Testing Methods

• Random testing - requires large numbers data permutations and combinations and can be inefficient
• Partition testing - reduces the number of test cases required to test a class
  o State-based partitioning - tests designed in way so that operations that cause state changes are tested separately from those that do not
  o Attribute-based partitioning - for each class attribute, operations are classified according to those that use the attribute, those that modify the attribute, and those that do not use or modify the attribute
  o Category-based partitioning - operations are categorized according to the function they perform: initialization, computation, query, termination

Inter-Class Test Case Design
Multiple class testing
1. For each client class use the list of class operators to generate random test sequences that send messages to other server classes
2. For each message generated determine the collaborator class and the corresponding server object operator
3. For each server class operator (invoked by a client object message) determine the message it transmits
4. For each message, determine the next level of operators that are invoked and incorporate them into the test sequence

Tests derived from behavior models
Test cases must cover all states in the state transition diagram
breadth first traversal of the state model can be used (test one transition at a time and only make use of previously tested transitions when testing a new transition)
test cases can also be derived to ensure that all behaviors for the class have been adequately exercised

Specialized Testing

Graphical User Interface (GUI) - test cases can be developed from behavioral model of user interface, use of automated testing tools is strongly recommended (see Chapter 12)
Client/Sever Architectures - operational profiles derived from usage scenarios are tested at three levels (client application "disconnected mode", client and server software without network, complete application)
Applications function tests
Server tests
Database tests
Transaction tests
Network communications tests
Documentation and Help
  - Review and inspection to check for editorial errors
  - Black-Box for live tests
    - Graph-based testing to describe program use
    - Equivalence partitioning and boundary value analysis to describe classes of input and interactions
Real-Time Systems
1. Task testing - test each task independently
2. Behavioral testing - using technique similar to equivalence partitioning of external event models created by automated tools
3. Intertask testing - testing for timing errors (e.g., synchronization and communication errors)
4. System testing - testing full system, especially the handling of Boolean events (interrupts), test cases based on state model and control specification

Testing Patterns
• Provide software engineers with useful advice as testing activities begin
• Provide a vocabulary for problem-solvers
• Can focus attention on forces behind a problem (when and why a solution applies)
• Encourage iterative thinking (each solution creates a new context in which problems can be solved)