Software Testing Overview & Techniques

1. Software Testing Fundamentals

Formal reviews by themselves cannot locate all software errors. Testing occurs late in the software development process and is the last chance to catch bugs prior to customer release.

Testability: "is simply how easily a computer program can be tested."

The following characteristics lead to testable software:

- **Operability**: it operates cleanly if implemented with quality in mind
- **Observability**: the results of each test case are readily observed. Variables are visible during execution. Source code is available.
- **Controllability:** the degree to which testing can be automated and optimized
- **Decomposability:** testing can be targeted; independent modules can be tested independently.
- **Simplicity:** reduce complex architecture and logic to simplify tests; code simplicity ➔ coding standards.
- **Stability**: few changes are requested during testing, the S/w recovers well from failures.
- **Understandability**: Changes to the design are communicated to the testers.

Test Characteristics

1. A good test has a high probability of finding an error. Tester must understand the system and develop a mental picture of how it might fail.
2. A good test is not redundant; every test should have a different purpose.
3. A good test should be "best of breed"; the test that has the highest likelihood of uncovering a whole class of errors should be used.
4. A good test should be neither too simple nor too complex.

Megha Gupta
Assistant Professor
Department of Computer Science

2. Black-Box and White-Box Testing

This section discusses the differences between black-box and white-box testing.

"Bugs lurk in corners and congregate at boundaries ..."

Black-Box testing alludes to tests that are conducted at the software interface. It examines some fundamental aspects of a system with little regard for the internal logical structure of the software.

White-Box testing is predicated on close examination of procedural detail. Logic paths through the S/W and collaborations between components are tested by providing test cases that exercise sets of conditions and/or loops.

3. White-Box Testing

This section makes the case that white-box testing is important, since there are many program defects (e.g. logic errors) that black-box testing can not uncover.

Sometimes called *glass-box testing*, is a test case design philosophy that uses the control structure described as part of component-level design to derive test cases.
The S/W eng. can derive test cases that:

1. Guarantee that all independent paths within a module have been exercised at least once,
2. Exercise all logical decisions on their true and false sides,
3. execute all loops at their boundaries and within their operational bounds,
4. Exercise internal data structures to ensure their validity.

Megha Gupta
Assistant Professor
Department of Computer Science

4. Basis Path Testing

This section describes basis path testing as an example of a white-box testing technique.

Basis Path Testing is a white-box testing technique which enables the test case designer to derive logical complexity measure of procedural design and use this measure as a guide for defining a set of execution paths.

Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least once time during testing.

4.1 Flow Graph Notation

A flowchart is used to depict program control structure. Each circle, called a *flow graph node*, represents one or more procedural statements. A sequence of boxes and decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links*, represent flow of control and are analogous to flowchart arrows.

An edge must terminate at a node, even if the node does not represent any procedural statements.

Areas bounded by edges and nodes are called *regions*. When counting regions, we include the area outside the graph as a region.

5. Control Structure Testing

Basis path testing is one form of control structure testing. This section introduces three others (condition testing, data flow testing, loop testing). The argument given for using these techniques is that they broaden the test coverage from that which is possible using basis path testing alone.

6. Black-Box Testing

The purpose of black-box testing is to devise a set of data inputs that fully exercise all functional requirements for a program. It is important to know that in black-box testing the test designer has no knowledge of algorithm implementation. The test cases are designed from the requirement statements directly, supplemented by the test designer's knowledge of defects that are likely to be present in modules of the type being tested.

Megha Gupta
Assistant Professor
Department of Computer Science

It is also called *behavioral testing*, which focuses on the functional requirements of the software.

Black-box testing attempts to find errors in the following categories:

1. incorrect or missing functions
2. interface errors
3. errors in data structures or external database access
4. behavior or performance errors
5. initialization and termination errors

Black-box testing tends to be applied during the later stage of testing. Tests are designed to answer the following 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?

7. OOT—Test Case Design

Berard [BER93] proposes the following approach:

1. Each test case should be uniquely identified and should be explicitly associated with the class to be 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 [BER94]:
    a. a list of specified states for the object that is to be tested
    b. a list of messages and operations that will be exercised as a consequence of the test
    c. a list of exceptions that may occur as the object is tested
    d. 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)

e. supplementary information that will aid in understanding or implementing the test.

8. **Testing 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.

9. OOT Methods: Random Testing at the Class Level

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)
- generate a variety of random (but valid) test sequences
  - exercise other (more complex) class instance life histories

9.1 OOT 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
  - categorize and test operations based on their ability to change the state of a class
- attribute-based partitioning

- categorize and test operations based on the attributes that they use

- category-based partitioning
  - categorize and test operations based on the generic function each performs

9.2 OOT 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 collaborator class and the corresponding operator in the server object.
- For each operator 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 operators that are invoked and incorporate these into the test sequence

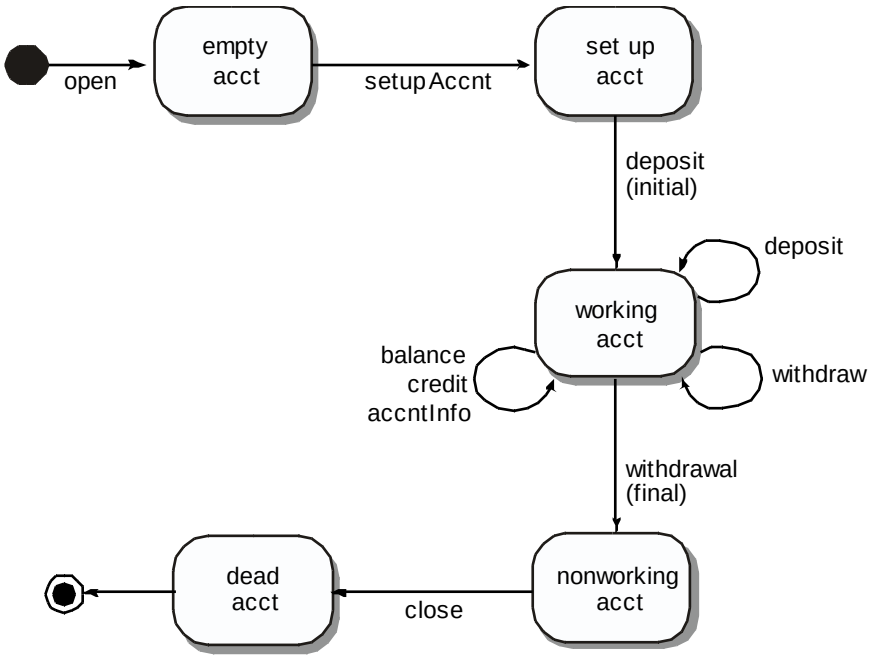

Figure 14.3 State diagram for Account class (adapted from [KIR94])

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