Software Quality Management

Software Testing Techniques

1

Fundamentals of Testing

- Testing conventional applications
  - White-box approaches
    - Basis path / condition / branch / data-flow
  - Black-box approaches
    - Equivalent class partitioning / boundary value analysis
- Testing object-oriented applications

2

The goal of testing is to find errors, and a good test is one that has a high probability

of finding an error. Therefore, you should design and implement a computer based

system or a product with “testability” in mind. At the same time, the tests

themselves must exhibit a set of characteristics that achieve the goal of finding the

most errors with a minimum of effort.

Object oriented testing (OOT)

- To adequately test OO systems, three things must be done:
  - 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.

3

The Test-Case Design Implications of OO Concepts

As a class evolves through the requirements and design models, it becomes a target

for test-case design. Because attributes and operations are encapsulated, testing

operations outside of the class is generally unproductive. Although encapsulation is

an essential design concept for OO, it can create a minor obstacle when testing.

Inheritance may also present you with additional challenges during test-case

design. I have already noted that each new usage context requires retesting, even

though reuse has been achieved. In addition, multiple inheritance4 complicates testing

further by increasing the number of contexts for which testing is required

If subclasses instantiated from a superclass are used within the same problem

domain, it is likely that the set of test cases derived for the superclass can be used

when testing the subclass. However, if the subclass is used in an entirely different context, the superclass test cases will have little applicability and a new set of tests must be designed.

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.

The object of *fault-based testing* within an OO system is to design tests that have a

high likelihood of uncovering plausible faults.

If real faults in an OO system are perceived to be implausible, then

this approach is really no better than any random testing technique. However, if the

analysis and design models can provide insight into what is likely to go wrong, then

fault-based testing can find significant numbers of errors with relatively low expenditures

of effort.

Inheritance does not obviate the need for thorough testing of all derived classes:

In sum, additional tests (and re-testing) will be needed for redefined (overloaded) operations and inherited operations that depend on those redefined ones.

Scenarios uncover interaction errors. But to accomplish this, test cases must be

more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test (users do not limit themselves to the use of one subsystem at a time).

Example:

Use Case: Print a New Copy

1. Open the document.
2. Select “Print” in the menu.
3. Check if you’re printing a page range; if so, click to print the entire document.
4. Click on the Print button.
5. Close the document.

But this scenario indicates a potential specification error. The editor does not do what the user reasonably expects it to do. Customers will often overlook the check noted in step 3. They will then be annoyed when they trot off to the printer and find one page when they wanted 100. Annoyed customers signal specification bugs.

You might miss this dependency as you design tests, but it is likely that the problem would surface during testing. You would then have to contend with the probable response, “That’s the way it’s supposed to work!”

OOT—Test Case Design

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:
   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.

It is necessary to test an OO system at a variety of different levels in an effort to uncover

errors that may occur as classes collaborate with one another and subsystems communicate

across architectural layers.

Test-case design methods for object-oriented software continue to evolve. However,

an overall approach toOOtest-case design has been suggested by Berard [Ber93]:

The white-box testing methods described in Chapter 18 can be applied to the operations

defined for a class. Basis path, loop testing, or data flow techniques can help to

ensure that every statement in an operation has been tested. However, the concise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

Black-box testing methods **are as appropriate for OO systems** as they are for

systems

developed using conventional software engineering methods. As I noted in

Chapter 18, **use cases** can provide useful input in the design of black-box and statebased

tests.

OOT Methods: Random Testing

- 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

Testing “in the small” focuses on a single class and the methods that are encapsulated

by the class. Random testing and partitioning are methods that can be used to

exercise a class during OO testing.

To provide brief illustrations of these methods, consider a banking application in

which an **Account** class has the following operations: *open(), setup(), deposit(), withdraw(),*

balance(), summarize(), creditLimit(), and *close()* [Kir94]. Each of these operations

may be applied for **Account,** but certain constraints (e.g., the account must be

opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem. Even with these constraints,

there are many permutations of the operations. The minimum behavioral life history

of an instance of **Account** includes the following operations:

open•setup•deposit•withdraw•close

This represents the minimum test sequence for account. However, a wide variety

of other behaviors may occur within this sequence:

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]n•withdraw•close

A variety of different operation sequences can be generated randomly. For example:

*Test case r*1*:* open•setup•deposit•deposit•balance•summarize•withdraw•close

*Test case r*2*:*
open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close

These and other random order tests are conducted to exercise different class instance life histories.

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

Tests are designed in a way that exercises operations

that change state and those that do not change state separately. Therefore,

Test case p1: open•setup•deposit•deposit•withdraw•withdraw•close

Test case p2: open•setup•deposit•summarize•creditLimit•withdraw•close

Test case *p*1 changes state, while test case *p*2 exercises operations that do not change state (other than those in the minimum test sequence).

For the **Account** class, the attributes balance and creditLimit can be used

to define partitions. Operations are divided into three partitions: (1) operations that

use creditLimit, (2) operations that modify creditLimit, and (3) operations that do not

use or modify creditLimit. Test sequences are then designed for each partition

operations in the **Account** class can be categorized

in initialization operations (*open, setup*), computational operations (*deposit, withdraw*), queries (*balance, summarize, creditLimit*), and termination operations (*close*).

OOT Methods: Inter-Class Testing

- Multiple-class testing
  - For each client class, use the list of class operations to generate a series of random test sequences. The operations 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 operation 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 operations that are invoked and incorporate these into the test sequence

Test-case design becomes more complicated as integration of the object-oriented system begins. It is at this stage that testing of collaborations between classes must begin. Like the testing of individual classes, **class collaboration testing** can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.

To illustrate [Kir94], consider a sequence of operations for the **Bank** class relative to an **ATM** class (Figure 19.2):

verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]n

A random test case for the **Bank** class might be

*Test case r*3 verifyAcct•verifyPIN•depositReq

In order to consider the collaborators involved in this test, the messages associated with each of the operations noted in test case *r3* are considered. **Bank** must collaborate with **ValidationInfo** to execute the *verifyAcct()* and *verifyPIN().* **Bank** must

collaborate with **Account** to execute *depositReq().* Hence, a new test case that exercises

these collaborations is

*Test case r*4 verifyAcct [Bank:validAcctValidationInfo]•verifyPIN

[Bank: validPinValidationInfo]•depositReq [Bank: depositaccount]

The approach for multiple class partition testing is similar to the approach used

for partition testing of individual classes. A single class is partitioned as discussed in

Section 19.5.2. However, the test sequence is expanded to include those operations

that are invoked via messages to collaborating classes.

OOT Methods: Behavior Testing

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

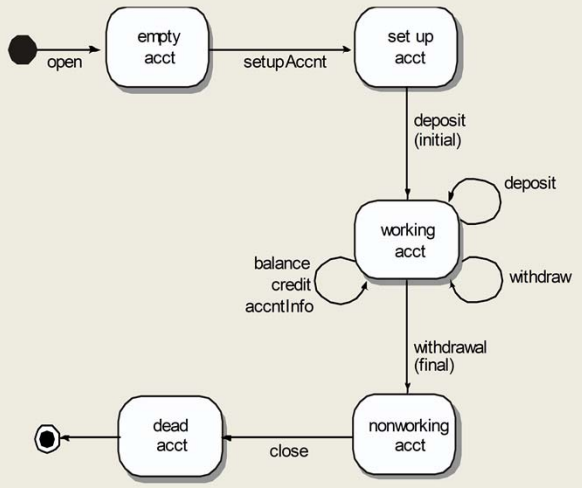

Figure 14.3 State diagram for Account class (adapted from [KIR94])

The state diagram for a class can be used to help

derive a sequence of tests that will exercise the dynamic behavior of the class (and

those classes that collaborate with it).

*Test case s*1*:* open•setupAccnt•deposit (initial)•withdraw (final)•close

It should be noted that this sequence is identical to the minimum test sequence

discussed in Section 19.5.2. Adding additional test sequences to the minimum

sequence,

*Test case s*2*:* open•setupAccnt•deposit(initial)•deposit•balance• credit•withdraw (final)•close

*Test case s*3*:* open•setupAccnt•deposit(initial)•deposit•withdraw•accntInfo•withdraw (final)•close

Still more test cases could be derived to ensure that all behaviors for the class

have been adequately exercised. In situations in which the class behavior results in

a collaboration with one or more classes, multiple state diagrams are used to track the behavioral flow of the system.

Summary

- OO testing inherits techniques from conventional testing, but also has its uniqueness.
  - Test shall be interpreted in a broader manner.
  - Focus on class behaviors and collaborations
    - Fault-based testing
    - Random testing
    - Collaboration testing
    - Behavioral testing