IRACST - International Journal of Computer Science and Information Technology & Security (IJCSITS), ISSN: 2249-9555
Vol. 2, No.2, April 2012

TESTING STRATEGIES FOR OBJECT ORIENTED SYSTEMS

Shalini Gambhir
Assistant Professor
I.T dept, Bharati Vidhyapeeth's College Of Engineering.
Delhi, India

***Abstract*—The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span. Although this fundamental objective remains unchanged for object-oriented software, the nature of OO programs changes both testing strategy and testing tactics.**

Keywords-;Object-orientation ;Test case;Testing Strategies ;)

I. INTRODUCTION

The architecture of object-oriented software results in a series of layered subsystems that encapsulate collaborating classes. Each of these system elements (subsystems and classes) performs functions that help to achieve system requirements. 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

II. OBJECT ORIENTED TESTING

A. Who does it?

Object-oriented testing is performed by software engineers and testing specialists.

B. Why is it important?

You have to execute the program before it gets to the customer with the specific intent of removing all errors, so that the customer will not experience the frustration associated with a poor-quality product. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

C. What are the steps?

OO testing is strategically similar to the testing of conventional systems, but it is tactically different. Because the OO analysis and design models are similar in structure and content
to the resultant OO program, "testing" begins with the review of these models. Once code has been generated, OO testing begins "in the small" with class testing. A series of tests are designed that exercise class operations and examine whether errors exist as one class collaborates with other classes.
As classes are integrated to form a subsystem, thread-based, use-based, and cluster testing, along with fault-based approaches, are applied to fully exercise collaborating classes.
Finally, use-cases (developed as part of the OO analysis model) are used to uncover errors at the software validation level.

D. What is the work product?

A set of test cases to exercise classes, their collaborations, and behaviors is designed and documented; expected results defined; and actual results recorded.

III. TESTING OOA AND OOD MODELS

Analysis and design models cannot be tested in the conventional sense, because they cannot be executed. However, formal technical reviews can be used to examine the correctness and consistency of both analysis and design models.

A. Correctness of OOA and OOD Models

The notation and syntax used to represent analysis and design models will be tied to the specific analysis and design method that is chosen for the project. Hence, syntactic correctness is judged on proper use of the symbology; each model is reviewed to ensure that proper modeling conventions have been maintained.
During analysis and design, semantic correctness must be judged based on the model's conformance to the real world problem domain. If the model accurately reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed), then it is semantically correct. To determine whether the model does, in fact, reflect the real world, it should be presented to problem domain experts, who will examine the class definitions and hierarchy for omissions and ambiguity. Class relationships (instance connections) are evaluated to determine whether they accurately reflect real world object connections.

B. Consistency of OOA and OOD Models

The consistency of OOA and OOD models may be judged by "considering the relationships among entities in the model. An inconsistent model has representations in one part that are not correctly reflected in other portions of the model".
To assess consistency, each class and its connections to other classes should be examined. The class-responsibility-collaboration model and an object-relationship diagram can be used to facilitate this activity. The CRC model is composed on CRC index cards. Each CRC card lists the class name, its

responsibilities (operations), and its collaborators (other classes to which it sends messages and on which it depends for the accomplishment of its responsibilities). The collaborations imply a series of relationships (i.e., connections) between classes of the OO system. The object-relationship model provides a graphic representation of the connections between classes. All of this information can be obtained from the OOA model.

To evaluate the class model the following steps have been recommended:

- ***Revisit the CRC model and the object-relationship model*** Cross check to ensure that all collaborations implied by the OOA model are properly represented.
- ***Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition***. For example, consider a class defined for a point-of-sale checkout system, called credit sale. This class has a CRC index card . For this collection of classes and collaborations, we ask whether a responsibility (e.g., read credit card) is accomplished if delegated to the named collaborator (credit card).
- ***Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source***. For example, if the credit card class receives a request for purchase amount from the credit sale class, there would be a problem. Credit card does not know the purchase amount.
- ***Using the inverted connections examined in step 3, determine whether other classes might be required and whether responsibilities are properly grouped among the classes.***

Class name: Credit sale	
Class type: Transaction event	
Class characteristics: Nontangible, atomic, sequential, permanent, guarded	
Responsibilities:	Collaborators:
Read credit card	Credit card
Get authorization	Credit authority
Post purchase amount	Product ticket
	Sales ledger
	Audit file
Generate bill	Bill

Figure1: An example of CRC index card used for review

- ***Using the inverted connections examined in step 3, determine whether other classes might be required and whether responsibilities are properly grouped among the classes.***
- ***Determine whether widely requested responsibilities might be combined into a single responsibility.*** For example, read credit card and get authorization occur in every situation. They might be combined into a validate credit request responsibility that incorporates getting the credit card number and gaining authorization.
- ***Steps 1 through 5 are applied iteratively to each class and through each evolution of the OOA model.***

IV. OBJECT-ORIENTED TESTING STRATEGIES

The classical strategy for testing computer software begins with "testing in the small" and works outward toward "testing in the large." In software testing, we begin with unit testing, then progress toward integration testing, and culminate with validation and system testing. In conventional applications, unit testing focuses on the smallest compliable program unit—the subprogram (e.g., module, subroutine, procedure, component). Once each of these units has been tested individually, it is integrated into a program structure while a series of regression tests are run to uncover errors due to interfacing between the modules and side effects caused by the addition of new units. Finally, the system as a whole is tested to ensure that errors in requirements are uncovered.

V. UNIT TESTING IN THE OO CONTEXT

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as *methods* or *services*) that manipulate these data. Rather than testing an individual module, the smallest testable unit is the encapsulated class or object. Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically. We can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class.
To illustrate, consider a class hierarchy in which an operation *X* is defined for the super class and is inherited by a number of subclasses. Each subclass uses operation *X*, but it is applied within the context of the private attributes and operations that have been defined for the subclass. Because the context in which operation *X* is used varies in subtle ways, it is necessary to test operation *X* in the context of each of the subclasses. This means that testing operation *X* in a vacuum (the traditional unit testing approach) is ineffective in the object-oriented context. Class testing for OO software is the equivalent of unit testing for conventional software.
Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

VI. INTEGRATION TESTING IN THE OO CONTEXT

Because object-oriented software does not have a hierarchical control structure, conventional top-down and bottom-up

integration strategies have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class".

There are two different strategies for integration testing of OO systems:

- The first, thread-based testing, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur.
- The second integration approach, use-based testing, begins the construction of the system by testing those classes (called independent classes) that use very few (if any) of server classes. After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed. Unlike conventional integration, the use of drivers and stubs as replacement operations is to be avoided, when possible. Cluster testing is one step in the integration testing of OO software.

Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

VII. VALIDATION TESTING IN AN OO CONTEXT

At the validation or system level, the details of class connections disappear. Like conventional validation, the validation of OO software focuses on user-visible actions and user-recognizable output from the system. To assist in the derivation of validation tests, the tester should draw upon the use-cases that are part of the analysis model. The use-case provides a scenario that has a high likelihood of uncovered errors in user interaction requirements. Conventional black-box testing methods can be used to drive validations tests. In addition, test cases may be derived from the object-behavior model and from event flow diagram created as part of OOA.

VIII. TEST CASE DESIGN FOR OO SOFTWARE

Test case design methods for OO software are still evolving. However, an overall approach to OO test case design has been defined by Berard :

1. Each test case should be uniquely identified and 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.

Unlike conventional test case design, which is driven by an input-process-output view?
of software or the algorithmic detail of individual modules, object-oriented testing focuses on designing appropriate sequences of operations to exercise the states of a class.

A. The Test Case Design Implications of OO Concepts

The OO class is the 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. , "Testing requires reporting on the concrete and abstract state of an object." Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

Inheritance also leads to additional challenges for the test case designer. We have already noted that each new context of usage requires retesting, even though reuse has been achieved. In addition, multiple inheritance3 complicates testing further by increasing or 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 super class test cases will have little applicability and a new set of tests must be designed.

B. Applicability of Conventional Test Case Design Methods

The white-box testing method 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 we noted earlier in this chapter, use-cases can provide useful input in the design of black-box and state-based tests.

C. Fault-Based Testing

The object of *fault-based testing* within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to customer requirements, the preliminary planning required to perform fault based testing begins with the analysis model. 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. Consider a simple example

Software engineers often make errors at the boundaries of a problem. For example, when testing a SQRT operation that returns errors for negative numbers, we know to try the boundaries: a negative number close to zero and zero itself. "Zero itself" checks whether the programmer made a mistake like

if (x > 0) calculate_the_square_root();

instead of the correct

if (x >= 0) calculate_the_square_root();

D. Testing Surface Structure and Deep Structure

Surface structure refers to the externally observable structure of an OO program. That is, the structure that is immediately obvious to an end-user. Rather than performing functions, the users of many OO systems may be given objects to manipulate in someway. But whatever the interface, tests are still based on user tasks. Capturing these tasks involves understanding, watching, and talking with representative users. For example, in a conventional system with a command-oriented interface, the user might use the list of all commands as a testing checklist. If no test scenarios existed to exercise a command, testing has likely overlooked some user tasks (or the interface has useless commands). In a object based interface, the tester might use the list of all objects as a testing checklist. The best tests are derived when the designer looks at the system in a new or unconventional way. For example, if the system or product has a command-based interface, more thorough tests will be derived if the test case designer pretends that operations are independent of objects.

Deep structure refers to the internal technical details of an OO program. That is, the structure that is understood by examining the design and/or code. Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the system and object design of OO software. The analysis and design models are used as the basis for deep structure testing.

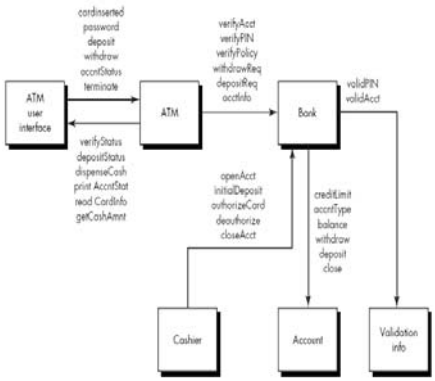

Figure 2: Example of object oriented model.

E. Interclass Test Case Design

Test case design becomes more complicated as integration of the OO system begins. It is at this stage that testing of collaborations between classes must begin. To illustrate "interclass test case generation" , lets take banking example :

The direction of the arrows in the figure indicates the direction of messages and the labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.

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.

F. Multiple Class Testing

Kirani and Tsai suggest the following sequence of steps to generate multiple

class random test cases:

1. 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.

2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.

3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.

4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

References:

[1] M. K.lling and J. Rosenberg, Blue - A Language for Teaching Object-Oriented Programming, in Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, 190-194, March 1996.

[2] Rosenberg, Linda, and Gallo, Albert, "Implementing Metrics for Object Oriented testing", Practical Software Measurement Symposium, 1999.