International Journal of Recent Engineering Science (IJRES),
ISSN: 2349-7157, *volume1 issue3 May to June 2014*

Specification-Based Class Testing – A Case Study

P.Kadambari , Dr.S.Prabu Anand

Assistant professor, Department of Computer Science
Ethiraj College for Women (Autonomous), Chennai

Abstract

Class testing is considered as the sole of the object-oriented software testing. In class testing we test each method, test relations among methods and tests inherit properties between every class and subclass. In specification-based testing we are mainly concerned about generating test cases from the specifications of the class. We never bother about testing the whole class at a time. It is not easy to manage the process of testing the class and to unify the test cases conveniently and consistently. In this paper, we will work how to give a structure to the test cases of the methods of a particular class under test. As per the given class specification, we will define the test cases and test suites. The paper will help in drawing the test cases and their further testing. We are using a new notation Object –Z to draw the structure and class specification.

1. Introduction

Software testing means the process of analyzing a software item to detect the difference between exiting and required conditions (that is, bugs) and to evaluate the features of the software item [1]. Software testing consumes a lot of time and cost of software development and maintenance. As the behavior and characteristics of software are expressed by the specification, so software testing should be started with a written and modeled specification. As specification is the reference and base of the testing, that is why name of specification-based testing [3].

According to the information gathered there are two types of software specifications- formal specifications and informal specification. Formal specifications are consistent, un-ambiguous and complete on the bases of mathematical semantics. So it provides favorable grounds for testing. In the structural testing field, practitioners have done a lot of work on formal specification-based testing. Model-based formal specification testing mainly based on the principle of partitioning the specification into equivalent classes and selecting small data from each class to verify the system behavior. Few proposed getting predicates from specification and then using constraint solver techniques to generate test cases. Some presented a domain partitioning method, to generate test cases.

The rest of paper is organizes are follows. Section 2 introduces the methods of class testing. Section 3 briefly introduces the Object-z specification language. Section 4 discusses the test method template in test class framework. Section 5 describes the finite state machine with respect to class testing. Section 6 introduces the test class framework in detail. Section 7 gives some conclusions and related work.

2. Class testing

The smallest unit for testing in procedural programs is a function or procedure. In object-oriented programs, this corresponds to the class member method. But, in object-oriented programs, methods are encapsulated in classes. So it is meaningless and difficult to test each method independently in object-oriented testing unless the relations among the methods of a class and their joint effect on shared states are also tested. So in object-oriented testing, the smallest unit for testing is a class.

Binder suggests that the class testing should include three aspects: testing methods, testing relation among methods and testing the inheritance features in a class [3]. The following steps should be followed for class testing. First, select adequate criteria for class testing. Second, select adequate criteria for method testing. Third, in method testing criteria, use category-partition, boundary analysis and other testing techniques to produce corresponding test cases for every method with in a class under test. Forth, use these test cases to construct test suites according to the class testing criteria. Fifth, use these test suites to test the class.

In specification-based testing, the class testing process means deriving test cases from the class method specification by the corresponding testing strategy and coverage analysis. Then class

specification is analyzed to construct the state transition graph for this class. In last, this state transition graph and some state-base coverage analysis techniques are used to generate test suites from the test cases for each method.

3. Object-Z

Object-Z [4] is an extension of the formal specification language Z to accommodate the object orientation. It helps in conducting individual operations with one state schema. All the definitions of state schema with associated operations give the definition of a *class*. In Object-Z specification of system are number of class definitions probably related by inheritance, a technique for class adaptation by modification or extension. Object-Z class syntax is shown below.

- **Visibility Lists**

Class's interface is defined in the visibility list. Visibility list clearly show which features- state variables, constants, initial state schema and operations are to be referred to in the environment of an object of class.

- **Inherited Classes**

If a class is derived from another class in Object-Z then its definitions- local definitions, state and initial state schemas, and operations- are merged with the parent class.

- **Local Definitions**

Types and constants which are used in the class are defined in the local definitions of a class.

- **State Schemas**

State variables of a class are defined in the state schema. State schema defines the possible states of the class.

- **Initial State Schemas**

Initial state of a class is defined in the initial state schema.

- **Operations**

The possible and permissible changes in the state that an object of the class may gain are defined in the operations.

Consider the given below Object-Z specification of the generic class Stack.

Class stack {	stck[++tos] = item;
int stck[] = new int[50];	}
int tos;	Int pop(){
stack() {tos = - 1;}	If (tos>0)
void push(int item){	Stct[tos--];
if(tos < 50-2)	}
	}

The class has a constant size of value 50. And one state variable p1. The state invariant stipulates that the size of the stack can not exceed SIZE. This class can be used further as the basis for defining the class in incremental order.

4. Test Method Template

Each method should be tested within class while testing a class. So method or function testing is essential and important part of class testing. A test method template defines an abstract test case for the method under testing. It defines the constraints of input variable and the corresponding expected output. Or the test method template defines test case set and Object-Z schema is used to represent the test method template.

4.1 Test Space

The valid input space (VIS) is defined with the preconditions of the method. It provides a set of inputs and state variables for stimulating the method/function. So Test Space is the source of test input data for the method. The valid output space (VOS) for a method is defined with the postcondition of this method. It provides the constraints of output and state variables for the method. Therefore, VOS is the source of expected output for the method. Test Space (TS) is considered to have VIS and VOS. It specifies the constraints of entering and exiting a method. It is the source of test case and can be divided to produce abstract test case- test method template (TMT). Thus, test space is also a test method template. VIS, VOS, TS and TMT can be defined as given below.

$VIS_{method =}$ pre Method

$VOS_{method =}$ post Method

$TS_{method =} VIS_{method} \wedge VOS_{method}$

$TMT_{method} == P\ TS_{method}$

In this paper, method name is used as subscript to represent the method under testing. *Pre* method and *post* method denote the precondition and post condition of the method respectively.

With the test space for each method, the test method template can be derived through the corresponding testing strategy.

4.2 Test Method Function

A strategy identifies a specific technique to draw test cases. While testing a method we can adopt many techniques at the same time. We can use domain testing, boundary value analysis, category partition and few other techniques those can use the specifications of the method. STRATEGY defines all the possible techniques for testing.

Test method function (TMF) is defined to manage the method testing process.

TMF_{method}:TMT_{method}XseqSTRATEGY→TMT_{method}

The test cases can be derived with the help of this function. Test method function, with the help of test space and test method template specifies the testing steps. TMF is used to specify the deriving process of concrete test data. As concrete test data is used as the instance of some TMT, the instance template (IT) is defined to represent the type of concrete test data. The given below formulas give the definition of instantiating a TMT.

$IT_{method} = TMT_{method}$

Instantiation: STRATEGY

$IT_{method} = TMF_{method}$ ($TMT_{method,}$,<instantiation>)

These definitions are applied in the method testing process for class STACK as follows:

Category partition testing

The category partition testing strategy is used to partition the test space for the PUSH method in the STACK class.

Category-partition: STRATEGY

$TMT_{push.\ 1} = [TS_{push} \mid \#\ item = 0]$

$\{TMT_{push.1}, TMT_{push,2}\} = TMF_{push}$ (TS_{push}, <category-partition>)

Boundary analysis testing

Test method template $TMT_{push,\ 2}$ and TS_{pop} are again partitioned with the help of boundary analysis testing.

| Boundary-analysis: STRATEGY

$TMT_{push\ 2.1} = [TMT_{push.2} \mid \#items = 1]$

$\{TMT_{Push.1}, TMT_{Push.2.1}, TMT_{Push.2.2}, TMT_{Push.2.3}\} = TMF_{Push}$ (TS_{Push}, < category-partition, boundary-analysis>)

$TMT_{Pop.1} = [TS_{Pop} \mid \#items = 1]$
$\{TMT_{Pop.1}, TMT_{Pop.2}, TMT_{Pop.3}, TMT_{Pop.4}\} = TMF_{Pop}$ (TS_{Pop}, <boundary-analysis>)

4.3 Method testing adequacy

It is never known in software testing when to stop testing which is big problem. Method testing adequacy defined for a tester to decide whether software has been tested adequately for a specific testing criterion. Here, structural coverage metrics is used to measure the thoroughness of a test set. Statement coverage, branch coverage and condition coverage are the traditional structural coverage metrics used to measure how well the bodies of each method have been tested. Testing adequacy function (MTAF) is defined as given below:

[CRITERIA] $|MTAF_{method}$: CRITERIA × TMT_{method} → TMT_{method}

Here, the type CRITERIA defines a set of all valid adequacy criteria. The method testing adequacy function (MTAF) estimates the set of test method template (TMT) based of some criterion and produces a set that satisfies this criterion. Branch coverage criterion is selected for the method push and pop to estimate the adequacy of the set of above test method templates. Then, the method testing adequacy functions of these methods are as follow;

$\{TMT_{push.1}, TMT_{push2.1}, TMT_{push2.2}, TMT_{push2.3},\} = MTAF_{push}$ (branch-coverage, $\{TMT_{push.1}, TMT_{push2.1}, TMT_{push2.2}, TMT_{push2.3},\}$)

{TMT $_{pop.1}$, TMT $_{pop.2}$, TMT $_{pop.3}$, TMT $_{pop.4}$,} = MTAF $_{pop}$ (branch-coverage, {TMT $_{pop.1}$, TMT $_{pop.2}$, TMT $_{pop.3}$, TMT $_{pop.4}$,})

5. Finite State Machine

Finite state machine model is widely used or object-oriented software testing. Interaction of methods with in a class and the class testing coverage can be drawn with this model. Finite state machine have finite and fixe number of states and input symbols. It consists of states, transitions, inputs and outputs. This section develops a finite state machine from the information of an Object-Z class specification.

5.1 State

The states of a class are represented with the values of state variables in Object-Z specification. State variables are defined in the state schema within the class. The INIT schema within a class states the initial state of the class. The methods in the class state the transitions from one state to another. So, a method is linked with to states – source state and target state. Method transforms source state into target state through some operation. So, source state is specified in the precondition of a method and target state is specified in the post condition of a method.

The test method templates (TMT) are derived from the precondition and post condition of each method of each method. TMT provide a partition of the source and target states. Hence, the states in finite state machine (FSM) of a class can be derived from the INIT schema within class and final test method templates. Steps to derive state in a finite state machine are shown below:

- INIT schema within the class represents the initial state of the class.
- Extract the precondition from each final test method template and hide the input variables in the precondition to represent the source state of some transition.
- Extract the post condition from each final test method template and hide the input/output variables in this post condition to represent the target state of some transition.
- Define each state using an Object-Z schema named state template (ST).

For *stack* class, the states of a FSM derived from the final TMT and INIT schema.

ST $_{INIT}$ = [items: seqT |items = <>]

ST $_{push.1.source}$ = [items: seqT |items = 0]

ST $_{pop.4.source}$ = [items seqT | #items = max]

ST $_{pop.4.target}$ = [items seqT | #items = max-1]

Few are equivalent states in the above states, which can be shown with one state.

State	Equivalent states
ST $_{INIT}$	ST$_{Push.1.shouce}$,ST$_{pop.1.target}$
ST $_{Push.1.1taget}$	ST$_{Push.2.1.shouce}$,ST$_{pop.1.source}$, ST $_{pop.2.target}$
ST $_{Push.2.1taget}$	ST$_{pop.2.source}$
ST$_{push.2.2source}$	ST $_{pop.3.target}$
ST $_{Push.2.2.taget}$	ST$_{pop.3.source}$
ST$_{push.2.3source}$	ST $_{pop.4.target}$
ST $_{Push.2.3.taget}$	ST$_{pop.4.source}$

Hence, states in the FSM of *stack* class are ST $_{INIT,}$ ST$_{Push.1.1taget}$, ST $_{Push.2.1taget}$, ST$_{push.2.2source}$, ST $_{Push.2.2.taget}$, ST$_{push.2.3source}$, ST $_{Push.2.3.taget.}$

Once the states of a FSM are derived, it is important to find out whether the states are disjoint. If not, any overlap of states must be resolved. To achieve this, the given below canonical disjunctive normal from (DNF) to construct a partition of state.

$$A \vee B = (A \wedge B) \text{ or } (\neg A \wedge B) \text{ or } (A \wedge \neg B)$$

With the help of this rule, we resolve the overlap in the above states can produce following disjoint states.

ST_0 = ST $_{INIT}$

ST_1 = ST $_{Push.1.1taget}$

ST_5 = ST $_{Push.2.3.taget}$

5.2 Transition

Transition means switching from one state to another in between two valid pair of states in FSM.

The steps of transitions in finite state machine shown below:

- Select a pair of states in FSM.
- Analyze the final test method template. If there exists a template that makes the source state in the state pair satisfy its precondition and the target state in the state pair satisfy its post condition, this test method template is used to label the transition of the state pair.
- Repeat above two steps until all valid transitions are labeled.

6. Test Class

Test class is the abstract representation of test suite. The interaction between the methods within a class is tested with the help of test class. A test class is represented with an Object-Z class. It has the test methods templates and INIT schema of the class under testing. The class testing adequacy criteria is concerned to construct a test suite.

Here, the state path coverage is used to build test suite. It refers to state tree, translated from the finite state machine. Leaf nodes show leaf states in the tree. Leaf state represents no further transition possible from this state. Test suites should cover each transition path, from initial state to leaf state.

6.1 Constructing State Tree

The initial state is transformed into root in the state tree. Transform each transition into an edge in the state tree. For one state, if there is no transition originates from it or it has been existed in the state tree, this state is labeled as a leaf state. Repeat step 2, 3 until all leaf states are labeled.

Searching this tress using depth first techniques, we gained given below paths. Each path is shown with a sequence.

$Path_1$ = <INIT>

$Path_2$ = <$TMT_{push.1}$, $TMT_{pop.1}$>

$Path_6$ =< $TMT_{push.1}$, $TMT_{push2.1}$, $TMT_{push.2.2}$, $TMT_{push.2.3}$, $TMT_{push.2.4}$, $TMT_{pop.4}$>

6.2 Constructing test class

A test class is represented as an Object-Z class. It states a test suite of the class under testing. The steps for constructing test classes of a class under testing are shown below:

- For each traverse (depth first) of the state tree of the class under testing following steps should be taken to construct a test class.
- TC_{class} notation is used to represent class under test. The subscript should be followed with a number representing the sequence number of test suite.
- State schema of the class under testing is used as the state schema of a test class.
- The INIT schema of the class under testing is used as the INIT schema of a test class.
- The functions of a test class are composed of all test method templates in the depth-first traversing path.
- Define a constant in the local definition of a test class representing the sequence of executing the test method templates in a depth-first traversing path.

  Following above shown steps the test classes of the *stack* class are shown below. The given type TMT represents the set of all test method templates for stack class.[TMT]

$TC_{stack.1}$	$TC_{stack.2}$
Max: N	Max: N
Testseq : seqTMT	Testseq : seqTMT
Testseq = <INIT>	Testseq=<$TMT_{push.1}$, $TMT_{pop.1}$>
Items : seq T	Items : seq T
#items < max	#items < max
INIT	INIT
Items = <>	Items = <>
	$TMT_{push.1}$, $TMT_{pop.1}$

$TC_{stack.3}$	$TC_{stack.6}$
Max: N	Max: N
Testseq : seqTMT	Testseq : seqTMT
Testseq= $TMT_{push.1}$, $TMT_{push.2.1,}TMT_{pop.2}$>	Testseq= $TMT_{push.1}$, $TMT_{push.2.1to2.3}$ $TMT_{pop.4}$>
Items : seq T	Items : seq T
#items < max	#items < max
INIT	INIT
Items = <>	Items = <>
$TMT_{push.1}$, $TMT_{push.2.1,}TMT_{pop.2}$	$TMT_{push.1}$, $TMT_{push.2.1,}$ $TMT_{push.2.2,}$ $TMT_{push.2.3,}TMT_{pop.4}$

Testing method of stack class is shown in detail in above test classes. These test class are showing the sequence of testing method with in the class and represent the information of test suits formally.

7. Conclusions

The case study has shown a class testing process. We used test method template, test method function, method testing adequacy function, and test class. In future our work will focus on the application of this technique on large classes.

References

[1] IEEE Std. 829-1998.

[2] Boris Beizer. Software Testing Techniques, Van Nostrand Reinhold, New York, 2nd ed., 1990.

[3] Robert V. Binder: Testing Object-Oriented System: Models, Patterns and tools. Addison Wesley Longman, Inc. 2000.

[4] Grame Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 2000, America.