Generating Test Cases for Object Oriented Programs Using Specification based Testing Techniques

Sujata Khatri

Deen Dayal Upadhyaya College, Delhi, India.

R.S.Chhillar

Department Of Computer Science & Applications, Rohtak, India.

Abstract:

In today's world software development industry and researchers has rapidly accepted the object-orientation paradigm for large scale system design. The object oriented language features of encapsulation, information hiding, inheritance, and polymorphism, promote concepts of modularity, reusability, and maintainability. If we are to capitalize on the potential advantages of object-orientation then it is important that the object-oriented approach is adopted and supported throughout the software development process. However the advantages of object-orientation cited above become potential disadvantages when we consider testing. Nowadays, it is clear that object-oriented paradigm, although provides many profitable features, requires special testing support and also provides opportunities for exploitation by a testing strategy. The majority of research is concerned with analysis, design and programming techniques. Software testing techniques have not kept pace with advances in object oriented system programming. The goal of this paper is to examine the testing of object oriented software and to derive test cases using equivalence partitioning and boundary value analysis technique for triangle problem.

Keywords: Object-oriented paradigm, Boundary value analysis, Equivalence partitioning.

1.1 Introduction and Overview:

Software applications are fastest growing trend in the virtual world and the possibilities regarding the features and functions provided by a specific application is generating tremendous interest amongst a vast number of people around the globe. As the interest grows, so does the demand for application. Since development of large software products involves several activities which are need to be suitably coordinated to meet desired requirements. Hence in today's world the importance of developing quality software is no longer an advantage but a necessary factor. Software testing is one the most powerful methods to improve the software quality directly. Testing is the process of finding differences between the expected (required) behaviour specified by system model and observed (existing) behaviour of implemented system. However complete (100 per cent) testing of a system is impossible as it must be performed under time and budget constraints**.** Software testing is difficult and expensive and testing object oriented system even more difficult.

1.2 Introduction to Object-oriented programming systems :

Meyer defines object-oriented design as "the construction of software systems as structured collections of abstract data type implementations" [2]. Object-orientation has rapidly become accepted as the preferred paradigm due to its features like structuring mechanism, information-hiding and software reuse for large scale system design [16]. However the advantages of object-orientation cited above become potential disadvantages when we consider testing. The structuring of the system as a set of independent classes requires that each of these must be tested and there may be a large number of them. In addition, information-hiding, which encourages designers of classes to have purely procedural interfaces, makes it difficult to determine whether the class is working correctly, since the state of internal data may not be accessible via the interface A tester often needs to spend significant time in developing lengthy testing code to ensure that system under test is reasonably well tested. To reduce the testing effort including time and budget becomes an important issue. Therefore how to persuade the designers to embed testing mechanism early at design phase to reduce testing cost is a new alternative in software testing strategy [11].The designers should consider how to make software which are more easily to be tested during software designing and thus cut the testing cost down in testing step.

The aim of this paper is to generate test cases using specification based testing techniques like equivalence partitioning and boundary value analysis for a C++ problem.

2. Related work:

Cheathem and Mellinger [3] suggested three levels of testing similar to conventional test level, although they called the second level as cluster testing rather than integration testing. Like stated earlier in this survey, they described that existing integration testing approaches are not sufficient for cluster level testing. Berard [4] addressed that object-oriented software does not have a “top”; therefore, integration testing approach based on tree hierarchies cannot be applied. Harrold et al. [5] presented an approach to decrease the effort in testing by reusing test cases from the superclass of the class under test. They suggested that each operation should be tested individually first, and then operations should be tested for interaction, both intra-class and inter-class, between them. This is similar to the concept proposed by Smith and Robson [6], who suggested that levels of testing should include algorithm level (individual operation), class level (inter-class operations), cluster level (intra-class operations), and system level. Johnson, who compiled a survey about object-oriented testing techniques during early 90’s [7], also agreed with this idea, although he expressed in the survey that there were some different ideas.

Jorgensen and Erickson suggested five levels of testing for object-oriented software [8]. They are method level, message quiescence level, event quiescence level, thread level, and thread interaction level. While it is obvious that method level is equivalent to unit testing in conventional testing, thread and thread interaction levels are equivalent to system testing. Unlike test level presented by others, test level presented by Jorgensen and Erickson provides seamless transition between levels of testing, as method testing turns into message-method path testing, and finally atomic system functions. However, there was a point made by several researchers [9, 10, 11] that basic unit for testing is a class, not an operation. Barbey and Strohmeier[12] addressed an issue of noninstantitable classes, as they cannot be instantiated for testing. From their concern, it is clear that the code under test must be tested as instantiated objects of the classes. In his report [Bin94] and later in his book [Bin99], Binder stated that a class is the basic unit for testing, as an operation is meaningless outside the context of its enclosing class. Other units Binder suggested are class cluster and system, which are similar in concept to

ones in [6, 7]. This idea becomes more acceptable later on, as researchers and practitioners find that it is more reasonable and more practical to test on class level than to test on operation level.

3. Introduction to Testing Techniques:

Tests can be derived from three different techniques: specification model, code and fault models. These techniques are not mutually exclusive but rather complementary. In this paper we mainly focussed on specification based testing or black box testing which derive test data from software specification. Tests are derived from requirement specification (for system testing), design specification (for integration testing) and detailed module specification (for unit testing).

Equivalence Partitioning: The basic idea of equivalence Partitioning is that testing is more effective if tests are distributed over all the domain of the program rather than concentrated mainly on some points.

Boundary Value Analysis: It is a special case of equivalence Partitioning technique. Values that lie at the edge of equivalence Partition are called boundary Value Analysis

3.1 Problem Statement:

Consider a C++ triangle program which accepts three sides of triangle as input. The output of the program is the type of the triangle determined by three sides: Equilateral, Isosceles, Scalene or Not a Triangle.

```
#include <iostream.h>
void main()
int side1,side2,side3,match=0;
cout<<"Enter length of side1"<<endl;
cin>>side1;
cout<<"Enter length of side2"<<endl;
cin>>side2;
cout<<"Enter length of side3"<<endl;
cin>>side3;
if(side1==side2)
        match=match+1;
elseif(side1==side3)
        match=match+2;
elseif(side2==side3)
        match=match+3;
if(match==0)
{
                If(side1+side2)<=side3
                        cout<<"not a triangle";
                else
                {
                        If(side1+side3)<=side2
                                cout<<"not a triangle";
                        else
                                cout <<"triangle is scalene";
                        }
        }
elseif(match==2)
```

```
        {
                If(side1+side3)<=side2
                        cout<<"not a triangle";
                else
                        cout<<"triangle is isosceles";


        }
elseif(match==3)
        {
                If(side2+side3)<=side1
                        cout<<"not a triangle";
                else
                        cout<<"triangle is isosceles";
else
        cout<<"triangle is equilateral";
        }
}
```

For the given problem we have generated test cases using Boundary Value Analysis and Equivalence class testing techniques.

3.1.1 Boundary value analysis technique:

It states the if we have n variables , we hold all but one at the nominal values and let the remaining variables assume the min, min+, nom, max-, max values, and repeat this for all variables.
Hence for a function of n variables, boundary value analysis results in 4n+1 test cases .
Test cases for the triangle problem: Since in the given problem there is no condition on the triangle sides, other than being integers. We take arbitrarily 1 as the lower bound and 100 as the upper bound.

Table 1.1

Test case 1	Side1	Side2	Side3	Expected output
1	1	50	50	Isosceles
2	2	50	50	Isosceles
3	50	50	50	Equilateral
4	99	50	50	Isosceles
5	100	50	50	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	50	50	Equilateral
9	50	99	50	Isosceles
10	50	100	50	Not a triangle
11	50	50	1	Isosceles
12	50	50	2	Isosceles
13	50	50	50	Equilateral
14	50	50	99	Isosceles
15	50	50	100	Not a triangle

And since test cases 3,8,13 are similar so finally we get only 13 test cases.

Table 1.2

Test case 1	Side1	Side2	Side3	Expected output
1	1	50	50	Isosceles
2	2	50	50	Isosceles
3	50	50	50	Equilateral
4	99	50	50	Isosceles
5	100	50	50	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a triangle
10	50	50	1	Isosceles
11	50	50	2	Isosceles
12	50	50	99	Isosceles
13	50	50	100	Not a triangle

3.1.2 Equivalence Class partitioning Technique: An Equivalence class is a group of data values where tester assumes that the test object processes them in the same way. For the given triangle problem, we note that there are four possible outputs: Scalene, Isosceles, Equilateral and Not a triangle. We can use the following classes to identify output (ranges) equivalence classes:

r1 = {<side1, side2, side3>: the triangles with sides side1, side2, side3 is equilateral}
r2 = {<side1, side2, side3>: the triangles with sides side1, side2, side3 is isosceles}
r3= {<side1, side2, side3>: the triangles with sides side1, side2, side3 is scalene}
r4= {<side1, side2, side3>: the triangles with sides side1, side2, side3 do not form a triangle}

These classes results in following test cases:

Table 2.1

Test cases	Side1	Side2	Side3	Expected output
TE1	2	2	2	Equilateral
TE2	5	5	6	Isosceles
TE3	4	5	6	Scalene
TE4	3	1	5	Not a triangle

However if we base equivalence classes on the input domain, we obtain a better set of test cases.

d1= {<side1, side2, side3>: the triangles with sides side1, side2, side3 are equal}
d2= {<side1, side2, side3>: the triangles with sides side1 and side2 are equal and side1 is not equal to side3}
d3={<side1, side2, side3>: the triangles with sides side1 and side3 are equal and side1 is not equal to side2}
d4={<side1, side2, side3>: the triangles with sides side2 and side3 are equal and side1 is not equal to side2}
d5= {<side1, side2, side3>: the triangles with sides side1,side2 and side3 are not equal }
d6= {<side1, side2, side3>:side1>side2+side3}
d7= {<side1, side2, side3>:side1=side2+side3}
d8= {<side1, side2, side3>:side2>side1+side3}
d9= {<side1, side2, side3>:side2=side1+side3}
d10= {<side1, side2, side3>:side3>side1+side2}
d11= {<side1, side2, side3>:side23side1+side2}

Test cases for Equivalence classes based on input domain:

Table 2.2

Test cases	Side1	Side2	Side3	Expected output
1	2	2	2	Equilateral
2	2	2	3	Isosceles
3	2	3	2	Isosceles
4	3	2	2	Isosceles
5	2	3	4	Scalene
6	5	2	2	Not a triangle
7	4	2	2	Not a triangle
8	2	5	2	Not a triangle
9	2	4	2	Not a triangle
10	2	2	5	Not a triangle
11	2	2	4	Not a triangle

4. Conclusion:

During the Whole software development life cycle of software, testing plays an important role. There are various issues related to the instrumentation of input program and symbolic testing of the program during testing. Instrumentation of input program and symbolic execution of the program are necessary steps to generate test cases. The aim of this paper is to generate test cases for object oriented programs using traditional testing techniques. Our observation is that boundary based techniques do not pay attention to data dependencies however the equivalence classes pay attention to data dependencies as it requires identification of the equivalence classes . Another observation is that functional testing techniques suffered from untested functionality and redundant tests as seen from the tables. Boundary value analysis should be done together with Equivalence class as faults are more likely discovered at the boundaries if equivalence classes. These techniques can be combined easily, but require enough freedom in selecting test data.

5. References:

[1] Berard, Edward V. Essays on Object-Oriented Software Engineering. Prentice-Hall, Englewood Cliffs, New Jersey, 1993, p. 10.

[2] Meyer, Bertrand. Object-oriented Software Construction. Prentice-Hall, New York, NY, 1988, p. 59, 62.

[3] T. J. Cheatham and L. Mellinger, “Testing Object-Oriented Software Systems”, Proceedings of the 1990 ACM Annual Conference on Cooperation, Washington, D.C., United States, 1990, pp. 161 –165.

[4] E. V. Berard, “Issues in the Testing of Object-Oriented Software”, Electro’94 International, IEEE Computer Society Press, 1994, pp. 211–219.

[5] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, “Incremental Testing of Object-Oriented Class Structures”, Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia, 1992, pp. 68 – 80.

[6] M. D. Smith and D. J. Robson, “A Framework for Testing Object-Oriented Programs”, Journal of Object-Oriented Programming, June 1992, pp. 45-53.

[7] M. S. Johnson, Jr., “A Survey of Testing Techniques for Object-Oriented Systems”, Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, 1996.

[8] P. C. Jorgensen and C. Erickson, “Object-Oriented Integration Testing”, Communications of the ACM, Volume 37, Issue 9, September 1994, pp. 30 – 38.

[9] S. Barbey and A. Strohmeier, “The Problematics of Testing Object-Oriented Software ”,Proceedings of SQM ’94 Second Conference on Software Quality Management, Edinburgh, Scotland, UK, Vol 11-426.

[10] R. V. Binder, “Testing Object-Oriented Systems: A Status Report”, American Programmer, April 1994.

[11] R. V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley , 1999.

[12] S. Barbey and A. Strohmeier, “The Problematic of Testing on Object-Oriented Software”, Proceedings of SQM ’94 Second Conference Software Quality Management, Edinburgh, Scotland, UK, Volume 2, 1994, pp. 411-426.

[13] [13]. M. J. Harrold and G. Rothermel, “Perform Data Flow Testing on Classes”, Proceedings of the 2nd ACM SIGSOFT Symposium of Foundations of Software Engineering, New Orleans, Louisiana.

[14] IEEE 729-1983, "Glossary of Software Engineering Terminology," September 23, 1982.

[15] Chung, Chi-Ming and Ming-Chi Lee. "Object-Oriented Programming Testing Methodology", published in Proceedings of the Fourth International Conference on Software Engineering and Knowledge Engineering, IEEE Computer Society Press, 15 - 20 June 1992, pp. 378 – 385.

[16] Michael Kolling Teaching Object Orientation with the Blue Environment Journal of Object Oriented Programming (1999) Volum12, Issue: 2, Publisher: SIGS Publications, Pages: 14–23.