BLACK BOX: A QUALITY IMPROVEMENT SOFTWARE TESTING METHOD

Ritu Agarwal[1], Qamar Alam[2] and Saoud Sarwar[3]

[1]Assistant Professor, VGI, Ghaziabad
Email: rituagarwal8585@gmail.com
[2]Lecturer, IMS, Roorkee
Email:qamaralam83@gmail.com
[3]HOD, AFSET, Faridabad
Email: saoud_mtechcs@yahoo.in

ABSTRACT

Testing is the precursor to debugging. Testing is commonly the forte of programmers and advanced users, and occurs when a product is new or is being updated and needs to be put through its paces to eliminate potential problems. Testing identifies "bugs" or imperfections so that they can be corrected in the debugging process, before the release of the product. Within the automated testing there are two methodologies: Black Box Testing and White Box Testing. In this paper we have compassed the different functional or black-box testing techniques like boundary value analysis, robustness, equivalence class testing etc. We have implemented this technique on well known cubic equation problem and evaluated various test cases in different testing techniques as a result.

***Keywords*:** Software, testing, Black-Box testing, test cases.

INTRODUCTION

It is the process used to help identify the correctness, completeness, security, and quality of developed computer software. Testing enhances the integrity of a system by detecting deviations in design and errors in the system. Testing is essential because of Software reliability, Software quality, System assurance, Optimum performance and capacity utilization, Price of non-conformance. There are two fundamental purposes of testing: verifying procurement specifications and managing risk. First, testing is about verifying that what was specified is what was delivered: it verifies that the product (system) meets the functional, performance, design, and implementation requirements identified in the procurement specifications. Second, testing is about managing risk for both the acquiring agency and the system's vendor/developer/integrator. In the Software Engineering "V" Model for ITS, testing is the first step in Integration and Recompostition. However, while testing is shown as one stage of the life cycle, it is important to understand that testing is also a continuous process within the life cycle. Testing begins with writing the requirements; each requirement must be written in a manner that allows it to be tested.

Software Testing

Overview

Testing software is operating the software under controlled conditions, to verify that it behaves "as specified", to detect errors, and to validate that what has been specified is what the user actually wanted. Here there are two types of testing techniques: Black Box and White Box Testing

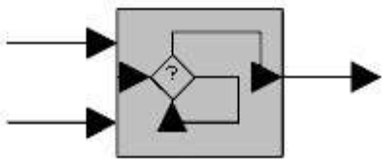
Black Box testing: This testing methodology looks at what are the available inputs for an application and what the expected outputs are that should result from each input. It is not concerned with the inner workings of the application.

White Box Testing: White box testing is performed based on the knowledge of *how* the system is implemented. The purpose of any security testing method is to ensure the robustness of a system in the face of malicious attacks or regular software failures. This testing includes analyzing data flow, control flow, information flow, coding practices, and exception and error handling within the system, to test the intended and unintended software behavior. It also called as clear box testing, glass box testing, transparent box testing, or structural testing.

Black Box versus White Box testing

Black-Box testing: Here, we don't know the exact content of the module we want to test. Consequently, we focus on the input-output behaviour which is (/should be) fixed by the specification. This can be done by any third person who hasn't developed the module and can also be automated.

White-Box testing: With further knowledge about the module, we can test in more detail. The input data can be chosen in a way that, for example, every statement is executed at least once. It is harder to do this, because you have to know the program-code very well. However, more errors can be found.



Experimental works

Here we have various methodologies in black box testing. We have evaluated various test cases according to the technique for cubic equation.

Boundary Value Analysis: It is observed that boundary points for any inputs are not tested properly. This leads to many errors. Large number of errors tends to occur at boundaries of

the input domain. Boundary Value Analysis (BVA) leads to selection of test cases that exercise boundary values. Here we have a cubic equation. We have to evaluate various outputs according to the various inputs. Experience shows that test cases that are close to boundary conditions have higher chances of detecting an error. Here boundary condition means an input value may be on the boundary or just above the boundary. Here we have input variables with in a limit of **0-100**. The boundary values are **0, 1, 50, 99, 100**. In this total number of tesst cases would be **(4n+1).** For a cubic equation the n would be 4, so total number of test cases would be

$4n+1= 4\times4+1= 17$ where n=4

Robustness Testing: Robustness is defined as the degree to which a system operates correctly in the presence of *exceptional inputs* or *stressful environmental conditions*. We can say it as an extension of BVA. Here we see that what happens when the extreme values are exceeded with a valve slightly greater than maximum and the value slightly less than minimum. There would be total number of test cases are **(6n+1)**. Here we have input variables with in a limit of **0-100**. The input variables are -1, 0, 1, 50, 99, 100, 101.For a cubic equation the n would be 4, so total number of test cases would be

$6n+1= 6\times4+1= 25$ where n=4

Equivalence Class Testing: Equivalence Partitioning Testing Method is a method used in Black Box Testing for the purpose of finding out minimum set of data for testing for valid and invalid inputs. A common approach is to divide the input domain into Equivalence Classes such that the program can reasonably be expected to behave the same for any points within a class. When dealing with predominantly numerical input space we can define the boundary of an equivalence class through the use of an algebraic inequality. In this method, the input domain of a program is partitioned into a finite number of equivalence classes such as the test of a representative value of each class is equivalent to a test of any other value. If one test case finds an error then other test cases in the class also would expect to find the same error. Conversely, if a test case did not find any error, we would expect that no other test cases in the class would find an error.

Two steps are required in implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid & invalid classes.
2. Generate the test cases using the equivalence classes identified previously. This is performed by writing test cases of valid equivalence classes.

In this paper we have a problem of Cubic Equation for which we have generated test cases using equivalence class testing technique. Here we have two domains: Input & output domain.

n=4 (a, b, c, d)

Input domain:

I_1 : {a : a=0}

I_2 : { a: a<0}

I_3 : { a:1<a<100}

I_4 : { a: a>100}

I_5 : { b: b<0}

I_6 : { b:1<b<100}

I_7 : { b: b>100}

I_8 : { c: c<0}

I_9 : { c:1<c<100}

I_{10} : { c: c>100}

I_{11} : { d: d<0}

I_{12} : { d:1<d<100}

I_{13}: { d: d>100}

Output Domain:

O_1 : { <a,b,c,d> : not a cubic equation if a=0}

O_2: {<a, b,c,d> : are real roots if x <0}

O_3 : { <a,b,c,d> : are equal roots if x= 0}

O_4 : { <a,b,c,d> : are imaginary roots if x<0}

Result tables

In this section we have summarized the results in the following tables. Here general cubic equation has the form ax^3+bx^2+cx+d the coefficients area, b, c, d the following cases need to be considered:

- If h> 0, then the equation has three distinct real roots.
- If h = 0, then the equation has a multiple root and all its roots are real.
- If h < 0, then the equation has imaginary root

Table 1. Bounded Value Analysis Result

Test Case	a	b	c	d	Expected out put
1	0	50	50	50	Not Cubic equation
2	1	50	50	50	Real roots
3	50	50	50	50	Real roots
4	99	50	50	50	Real roots
5	100	50	50	50	Real roots
6	50	0	50	50	Real roots

Table 1. Bounded Value Analysis Result (Contd…)

7	50	1	50	50	Real roots
8	50	99	50	50	Imaginary roots
9	50	100	50	50	Imaginary roots
10	50	50	0	50	Real roots
11	50	50	1	50	Real roots
12	50	50	99	50	Imaginary roots
13	50	50	100	50	Imaginary roots
14	50	50	`50	0	Imaginary roots
15	50	50	50	1	Equal roots
16	50	50	50	99	Real roots
17	50	50	50	100	Real roots

Note: Total no of test cases= 4n+1 where n =4

Table 2. Robustness Test cases Result

Test Case	***a***	***b***	***C***	***d***	***Expected out put***
1	-1	50	50	50	Invalid input
2	0	50	50	50	Not Cubic equation
3	1	50	50	50	One Imag & two Real roots
4	50	50	50	50	One Imag & two Real roots
5	99	50	50	50	One Imag & two Real roots
6	100	50	50	50	One Imag & two Real roots
7	101	50	50	50	Invalid input
8	50	-1	50	50	Invalid input
9	50	0	50	50	One Imag & two Real roots
10	50	1	50	50	One Imag & two Real roots
11	50	99	50	50	Imaginary roots
12	50	100	50	50	Imaginary roots
13	50	101	50	50	Invalid input
14	50	50	-1	50	Invalid input
15	50	50	0	50	One Imag & two Real roots
16	50	50	1	50	One Imag & two Real roots
17	50	50	99	50	Imaginary roots
18	50	50	100	50	Imaginary roots
19	50	50	101	50	Invalid input
20	50	50	50	-1	Invalid input
21	50	50	50	0	Imaginary roots
22	50	50	50	1	Imaginary roots

Table 2. Robustness Test cases Result (Contd…)

23	50	50	50	99	One Imag & two Real roots
24	50	50	50	100	One Imag & two Real roots
25	50	50	50	101	Invalid input

Note: Total no of test cases= 6n+1 where n =4

Table 3. Equivalence class Testing Results

a	b	c	d	Expected o/p
0	50	50	50	Not cubic Equation
-1	50	50	50	Imaginary root
50	50	50	50	Real root
99	50	50	50	Real root
100	50	50	50	Real root
101	50	50	50	Real root
50	0	50	50	Real root
50	-1	50	50	Real root
50	50	50	50	Real root
50	99	50	50	Imaginary root
50	100	50	50	Imaginary root
50	101	50	50	Imaginary root
50	50	0	50	Real root
50	50	-1	50	Real root
50	50	50	50	Real root
50	50	99	50	Imaginary root
50	50	100	50	Imaginary root
50	50	101	50	Imaginary root
50	50	50	0	Imaginary root
50	50	50	-1	Imaginary root
50	50	50	50	Real root
50	50	50	99	Real root
50	50	50	100	Real root
50	50	50	101	Real root

Result of experiments

Type of Testing	Test Case	Performance
Boundary value	17	Good
Robustness Testing	25	Fair
Worst Testing	625	Worst

COCLUSION

Testing is an important technique for the improvement and measurement of a software system's quality. It is a good idea to design a black-box test at the same time you write the program. This reveals unclarities and points in the problem statement, so that you can take them into account while writing the program instead of having to fix the program later. This paper has investigated the problem of cubic equation in order to detect domain faults. We have shown a purely geometric basis for test generation. In this we investigate various possible inputs and generate the desired outputs with the help of various black box testing methodologies.

CODE

```
#include<stdio.h>
  #include<math.h>
  int main()
  {
   char order;
   double E;
   int i=0,s=1,r=1;
   double N = 12;
   double a,b,c,d;
   double fx,fpx,X0,x;
   printf("Enter the value of a, b, c and d: ");
   scanf("%lf %lf %lf %lf",&a,&b,&c,&d);
   if(a>0)
   {
   printf("Enter the value of X0 (initial guess): ");
   scanf("%lf",&X0);
   printf("Enter the value of convergence limit: ");
   scanf("%lf",&E);
   while(order!='q')
   {
    x= X0;
   fx = a*x*x*x +b*x*x +c*x +d;
   i=0;
```

```
 while(i<N)
 {
 i=i+1;
 fx = a*x*x*x +b*x*x +c*x +d;
 fpx = 3*a*x*x +2*b*x +c;
 x= x - (fx/fpx);
 printf("After iteration %d, root = %f\n",r++,x);
 }
 printf("Press 'c' to continue or 'q' to quit: ");
 scanf("%c",&order); if(order=='c')
 printf("Enter the value of a, b, c and d (separated by space): ");
 scanf("%lf %lf %lf %lf",&a,&b,&c,&d);
 }
 }
}
```

FUTURE ASPECTS

The future approach for us is to implement the program code for calculating the various roots for any cubic equation according to any input and would be test that code with the help of White box test methodology.

REFERENCES

1. Amla, N. and Ammann, P. 1992. Using Z specifications in category partition testing. In COMPASS 92, Seventh Annual Conference on Computer Assurance. Gaithersburg, MD, USA,
2. Clarke, L. A., Hassell, J., and Richardson, D. J. 1982. A close look at domain testing. IEEE Transactions on Software Engineering 8, 4, 380–390.
3. White, L. J. and Cohen, E. I. 1980. A domain strategy for computer program testing. IEEETransactions on Software Engineering 6, 3, 247–257. ACM
4. Grochtmann, M. and Grimm, K. 1993. Classification trees for partition testing. Journal of Software Testing, Verification and Reliability 3, 2, 63–82.
5. Michael, C. C., McGraw, G., and Schatz, M. A. 2001. Generating software test data byevolution. IEEE Transactions on Software Engineering 27, 12, 1085–1110.ACM Journal Name, Vol. V, No. N, Month 20YY.
6. Beizer, B. Software Testing Techniques 2nd Edition.Van No strand Reinhold, 1990.

7. Lyu, M.R. (Ed.), Handbook of Software Reliability Engineering, Mc-Graw-Hill/IEEE, 1996.
8. Perry, W. Effective Methods for Software Testing, Wiley, 1995.
9. Poston, R.M. Automating Specification-based Software Testing, IEEE, 1996.
10. Howden, W.E., Reliability of the Path Analysis Testing Strategy. IEEE Transactions on Software Engineering, 2, 3, (Sept. 1976) 208-215
11. Morell, L.J. A Theory of Fault-Based Testing. IEEETransactions on Software Engineering 16, August 1990), 844-857.
12. Ostrand, T.J., and Balcer, M. J. The Category-Partition Method for Specifying and Generating Functional Tests. Communications of ACM, 31, 3(June 1988), 676-686.
13. Weyuker, E.J. On Testing Non-testable Programs. The Computer Journal, 25, 4, (1982) 465 470
14. Wakid, S.A., Kuhn D.R., and Wallace, D.R. Toward Credible IT Testing and Certification, IEEE Software, (August 1999) 39-47.
15. Weyuker, E.J., Weiss, S.N, and Hamlet, D. Comparison of Program Test Strategies in Proc.Symposium on Testing, Analysis and Verification TAV 4 (Victoria, British Columbia, October 1991), ACM
16. [Hamlet94] Dick Hamlet; Foundations of software testing: dependability theory; Proceedings of the second ACM SIGSOFT symposium on Foundations of software engineering , 1994, Pages 128 – 139
17. Precursor work of Ballista project. Tests selected POSIX functions for robustness.
18. Hetzel, William C., The Complete Guide to Software Testing, 2nd ed. Publication info: Wellesley, Mass.: QED Information Sciences, 1988. ISBN: 0894352423.Physical description: ix, 280 p.: ill ; 24 cm.
19. A standard for testing application software, William E. Perry, 1990
20. Yang, M.C.K.; Chao, A. Reliability-estimation and stopping-rules for software testing, based on repeated appearances of bugs; IEEE Transactions on Reliability, vol.44, no.2, p. 315-21, 1995