Prioritizing Test Case Generation for Software Testing in High Level Programming Development Environment

Dinesh Kumar Saini
Faculty of Computing and IT,
Sohar University, Oman
Faculty of Engineering and IT,
University of Queensland,
Brisbane, Australia

ABSTRACT
Abstract—Software testing is very tedious process and during the software development, testing needs time, efforts and money. Testing and retesting is part of development process and lots of efforts are needed for doing this. Detecting faults and errors in the early stages of development is the main task of any testing team. The entire test suits are written for the same target and the test suits grows as the software evolves over the period of time. Resources are very limited and due to resource constraints like cost, time and money, it is advised to prioritize the execution of test cases so that it can increase chances of early detection of faults in the software development process [1]. In this paper, high level language programming paradigm is considered for the development environment and algorithmic approach of design is considered. In this paper we present a new approach to prioritize test cases of particular software based on the requirements given by the client using high level functional programming language. Running all test cases in a normal Test suite, however, can consume an inordinate amount of time so, its main purpose is to improve rate of fault detection by prioritizing the test cases in a very short span of time and release the updated software to the customer [2]. In this paper a new test case prioritization algorithm, which calculates using data mining technique K-Nearest neighbor, which in turn uses Euclidean distance method approach to prioritize the test cases is proposed.

General Terms
Software Systems, Software Engineering and Software Testing

Keywords
Data Mining, Test Case, Prioritization, Software development, Time, Cost and Efforts

1. INTRODUCTION
Software Engineering [1] is the establishment and use of sound engineering principles in order to obtain economically reliable and efficiently developed software. It is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches. Software testing is an important and expensive stage in the software development life cycle. The software testing starts even at the stage of software architecture selection. As software changes over time, test suites are developed and used to test the modified software to make sure that changes do not affect the existing functionality in unintended ways, and to test for new functionality.

Due to time and resource constraints, it may not be possible to continually execute all the tests in suites, in testing iteration process. It is therefore important to prioritize (order) the execution of test cases in test suites so as to execute those test cases early on during testing, whose output is more likely to change. If prioritization can be achieved in the testing process it will help in early detection of errors and fault. Various techniques are addressed in the paper that can be used in test case prioritization. In this paper a new approach for prioritizing the execution of existing test cases with the goal of early detection of faults in the testing process is discussed in detail using high level programming paradigm [3].

In this paper we mainly concentrate on prioritizing the test cases of the software which has already been developed and is seeking modifications or further up gradations to the existing software. The test cases and the faults corresponding to each module are initially recorded by the programmer when the software is developed. Later, when the client approaches for any modification or up gradations to the prevailing software, test suits must be revisited and all the new test cases needed for the new updates is to be included. The modules in which the modifications are made are the one which we are concerned for testing [4,5].

So, consider the faults with respect to that module from the database and also all the test cases with respect to their faults. By correlating the faults, using the coverage of the faults and using the data mining technique k-nearest neighbor [15] (Euclidean distance approach) we arrive at prioritizing the test cases.

So, this new method of prioritizing the test cases prioritizes them based on the Euclidean distance between the module which is modified and all the test cases with respect to the faults (which are generated by corresponding modules). The test case with minimum distance is the most prioritized test case.

The faults for each test case are initially given by the developer and this is used while calculating the Euclidean distance. The faults with respect to each module are also given by the developer. The weight factor is calculated by using the line coverage (i.e. line at where fault occurred to that of function length). The remainder of this paper is organized as follows.

The background for the work is explained in section 3.Introduction to Data mining and its related techniques are explained in Section 4. Proposed algorithm is mentioned in section 5.Case study is discussed in Section 6. The experimental study, along with results and discussion, analysis are given in Sections (7, 8) and conclusions are given in Section 9.

2. PROBLEM STATEMENT

Some software was developed and tested and is currently in use .Customer requires some more features and informs to developer. The time constraint (i.e. time to update the software is less) .Now instead of testing all test cases which were initially tested part of them are selected by prioritizing in the method described below.

According to Rothermel at el. [5, 9] defines the test case

Prioritization problem as follows:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers

Problem: Find T' belongs to PT such that (for all T") (T" belongs to PT) (T" ≠ T') [f (T') ≥ f(T")].

Here, PT represents the set of all possible prioritizations (orderings) of T and f is a function that, applied to any such ordering, yields an award value for that ordering [2,7].

The objective of this research is to develop a test case prioritization technique that prioritizes test cases on the basis of detection of fault rat

2.1. Software System and Failure Cause

Failure causes can be tested using the testability concept which can be quantified to some extent; failure is smaller part of the big software, which is shown in the figure1.

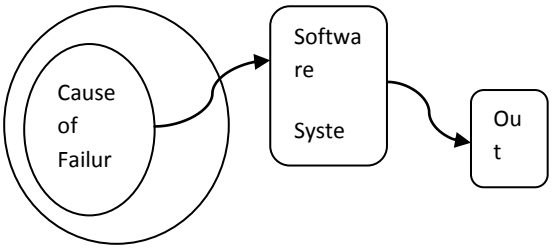

Figure1

Testability = FC/I*100%, FC= Failure Cause and I=Input

Testability is very difficult to Measure

3. BACKGROUND

An early work on test case prioritization by Elbaum [9] and Rothermel, [20,21] using the fault detection rate, is a measure of how quickly faults are detected during the testing process. These test case prioritization algorithms calculates average faults found per minute and it states that improved rate of fault detection during regression testing can provide faster feedback on a system under regression test. Horgan and London [12] present a new greedy heuristic algorithm for selecting a minimal subset of test suite T which covers all the requirements. Aggarwal et al.[14] describe a coverage based technique for test prioritization , where prioritization is based on the original test suite used for testing P and not one for testing modified version P'. However they don't combine code coverage information with function coverage. The other study by Elbaum, Malishvesky, Rothermel [5] presents an approach to prioritize test cases based on the coverage requirements present in the relevant slices of the outputs of test cases. Kim and Porter [12] propose a history based technique while Srivastava and Thiagarajan [17] have reported Echelon, a tool used to prioritize test. Srikanth and Williams investigate economic application of test prioritization [19] and Doet al. performed a cost benefit analysis of prioritizing JUnit test cases [13]. Since code and function coverage techniques are applied separately but yet they are not combined with each other to get better results or to perform new experiments. The general algorithm for prioritizing regression test cases based on functional coverage as explained in [9].

But, unlike all the above test case prioritizations this approach concentrates on both code and function coverage combinable. The term weight factor has been introduced which is based on the line at where fault occurred in the code and function length. This new approach based on this weight factor and k-Nearest Neighbor using Euclidean distance prioritizes the test cases are explained in the next sections.

4. INTRODUCTION TO DATA MINING

Data mining techniques are emerging a powerful new technology for the software development environment with great potential to help companies focus on the most important information in their testing databases which are produced by the testing teams.

Using data mining tools, software development teams can predict future trends and behaviours in the detection of errors and faults in the development environment. The data mining techniques will allow software testing businesses to make proactive, knowledge-driven decisions that will help the software development industry. The automated, prospective analyses offered by data mining move beyond the analyses of past events provided by retrospective tools typical of decision support systems in the software development environment. Data mining tools helps to answer quality and reliability questions that traditionally were too time consuming to resolve in the software development environment. Data mining helps in scour databases for hidden patterns. In testing environment finding predictive information that testing experts may miss because it lies outside their expectations can also be traced [10].

The most commonly used technique of data mining that is used in this approach for test case prioritization is

4.1 K-Nearest Neighbour (*k*-NN)

Objects can be classified based on closest training examples in the given feature space. *K*-NN is a type of instance-based learning, or lazy learning where the function is only approximated locally and all computation is deferred until classification. It can also be used for regression

This method will help in this case like, after finding the distances between the test cases and the modules with respect to faults, the nearest distance test case is more prioritized one.

4.2 Euclidean distance:

In mathematics, the Euclidean distance or Euclidean metric is the "ordinary" distance between two points that one would measure with a ruler, which can be proven by repeated application of the Pythagorean theorem. By using this formula as distance, Euclidean space becomes a metric space

The Euclidean distance between points $P = (p_1, p_2, \ldots, p_n)$ and $Q = (q_1, q_2, \ldots, q_n)$ in Euclidean *n*-space, is defined as:

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}.$$

This Euclidean method is useful for calculating the distances between the test cases and the modules with respect to faults as explained in the next section [10].

5. PROPOSED ALGORITHM

The methodology for implementing the algorithm is that, initially when the software was developed and tested, the faults corresponding to each module and its functions, line where the fault is introduced in the code and length of function in which it is found are stored in the Fault-module table as shown in table.1 and also the test cases and its detected faults are stored in the Fault-test case table as shown in table.2.

Later, when the customer approaches for any modification in the prevailing software, the modules corresponding to the changes are taken in to consideration and from the Fault module table which was initially stored in the database, the Fault-module array shown below is filled like if the fault is present among the faults corresponding to the modules under modification then the value is 1 otherwise it is 0.

After obtaining the array, the Euclidean distance to each of the test cases with respect to faults is calculated. Even weight factor is included in the calculation. The test case whose distance is less is the more prioritized test case.

Table.1 Fault-module

Fault _id	Module _id	Function _id	line_where _fault_inroduced	function _length

Table.2 Fault-testcase

Test_id	F1	F2	F3	F4	F5

This array consists of binary values corresponding to each fault that are present in the modules under modification. If the fault is present among the faults corresponding to the modules under modification then the value is 1 otherwise it is 0.

Fault_module[n]

F1	F2	F3	F4	F5

This array consists of calculated weighted factor for each element in Fault_module[] array using the Fault-module table. The weight factor for each fault is calculated as the ratio of line where it is introduced in the code and function length in which it is present.

Weight_factor[n]

F1	F2	F3	F4	F5

5.1 Algorithm for Calculating Euclidean Distance

```
Void compute Euclidean_distance( )
{
        int final[];
j=0;
        for each Tj in Fault-testcase do
//here Tj represents each row
 in Fault-testcase table
                int sum=0, Euclidean-distance;
                for i=0 -->(n-1) do
                if(Fault_module[i]==1)

Sum=sum+[{Fault_module[i]–Tj[i]}^2* Weight_factor[i] ]
                        end if

 end for
                Euclidean-distance=(sum)^1/2 ;
                Final[j]= Euclidean-distance;
        end for
}
```

The algorithm is implemented using the formula of Euclidean distance by multiplying each term in the square root by the weight factor of corresponding fault. Looping is used to calculate distance to each of the test cases. For this test case table is used.

5.2 Algorithms for calculating weight factor:

```
Void compute Weight_factor ( )
{
        int Weight_factor[n], line_where
_fault_inroduced[n], function_length[n];

/* here function length and line_where_fault_inroduced  are
obtained
from the Fault-module table */

        for i=0 -->(n-1) do
        {
        Weight_factor[i] =( line_where
_fault_inroduced[i])/function_length[i];
        }
        end for
```

This algorithm is used for calculating the weight factor with help of fault-module table and storing it in the array named weight_factor[n]. For each fault it is calculated as the ratio of line where fault introduced and function length.

Clearly with this approach it is ensured that

1. All test cases that are initially available are considered for implementing this process. No fault is neglected.
2. We have used the weight factor for prioritizing the faults with respect to the function coverage and code coverage.
3. This is an efficient method for prioritizing the test cases as it is considering all faults with respect to modules and test cases and also the code coverage and function coverage.

When modules and functions under modification are known and suppose 'n' number of faults are detected. Consider there are m number of test cases then it takes O(m*n) to compute distances for all test cases and it takes O(log(m)) to sort the distances .Hence total time complexity is O(m*n).

6. CASE STUDY

The above algorithm is demonstrated with the help of the below mentioned example. There are 2 modules given , the first module contains two functions namely main( ) and factorial( ). The second modules also contains two functions namely add( ) and divide( ) functions. The lines highlighted in bold letters in the code mentioned above signify the lines where the fault is introduced.

MODULE 1:

```
int main( )

1. {    printf("Enter operation: ");
2.      scanf("%C",&op);
3.      if (op==+)
4       {
5.              printf("Enter argument 1: ");
6.              scanf("%d",&arg1);
7.              printf("Enter argument 2: ");
8.              scanf("%d",&arg2);
                If(arg2>3)
                {
9.              add(arg1,arg2/0); //add(arg1,arg2)
                }
                Else
                {
                add(arg1,arg2)
                Div(arg1,arg2)
                }
10.     }

11.     else if(op=='/')
12.     {
13.             printf("Enter argument 1: ");
14.             scanf("%d",&arg1);
15.             printf("Enter argument 2: ");
16.             scanf(" %d,&arg2);
        //input(/,4,0);
        //fault is syntax error '' missing
17.             add(arg1,arg2);
        div(arg1,arg2);
18.     }

19.     else if(op='f')//input(f,2)
                //fault is op='f ' instead of op=='f'
20.     {
21.             printf("Enter argument 1: ");
22.             scanf("%d",&arg1);
23.             Factorial(arg1);
24.     }

25.     else
26.     {
27.             printf("invalid input!!");
28.     }
29.     }


void factorial(int x)
1.{
2.      int i;
3.      long int fact=1;

4.      if(x>0)
5.              {
6.              for(i=1;i<=x;i++)
7.              fact*==L;//instead of fact*=1;
8.              printf("%ld",fact);

9.              }
10.     else
11.     add(fact,2)
printf("Invalid argument");
12.     }
```

MODULE 2:

```
void add(int arg1, int arg2)
1.{
2.      int sum;
3.      sum = =arg1+ arg2;
// instead of sum=arg1+arg2;
4.      printf("Result is %d",sum);
        factorial('f',arg2);
        div(arg1,arg2);
5. }

void divide(int arg1, int arg2)
1.      {
2.      if(arg2==0)
3.      {
4.              printf("Invalid argument! ");
5.      }

6.      else
7.              printf("Result is %d",arg2/arg1);
8.      }
```

6.1 Faults introduced

In the above code fault occurrences were written in quotes or highlighted in bold letters and test cases which detect them are given below in table.1. Here in this the code was divided into two modules with each module containing two functions in it.

Table.3 Faults introduced

Fault Test Case	Function	Description	Detected by
F1	Add T1('+', 2 ,2)F4,F5	Sum==Arg1+arg2	
F2	Mam F4,F4	add (arg1, arg2/0) // instead of add (arg1, arg2)	T2(+,1,4)
F3	Mam T3('/',4,0), F5	missing double quotes	
F4	Factorial	fact*==L; //instead of fact*=1;	T4('f',3)
F5	Divide	arg2/arg1 instead of arg1/arg2	T5 (/,1,2)

From the above table it can be seen for T1 ('+',2,2) fault is occurring in the add function (F1) and it is calling factorial function and divide function in the add function which in turn leads to occurrence of faults F4 and F5.

Similarly for other test cases the faults are detected which are shown in below table.

Table.4 Test cases-Faults detected

Test cases	Faults Detected
1	F1, F4, F5
2	F2, F4, F5
3	F3, F5
4	F4
5	F5

Here in table.5 the modules and its corresponding faults, functions, line where fault occurred and function length are given below.

Table.5 Fault-module

Fault _id	Module _id	Function _id	line_where _fault_inroduced	function _length
1	2	1	3	5
2	1	1	19	30
3	1	1	6	30
4	1	2	7	12
5	2	2	7	7

Here in table.6 we have the table for test cases and faults from table .4. If the fault is detected the binary value is 1 otherwise it is 0.

Table.6 Fault-testcase

Test_id	F1	F2	F3	F4	F5
1	1	0	0	1	1
2	0	1	0	1	1
3	0	0	1	0	1
4	0	0	0	1	0
5	0	1	0	0	1

7. EXPERIMENTAL RESULTS

Consider upon customers request the modification is in function 1(i.e. main) function 2(i.e. factorial) in MODULE 1 and function 2 (i.e. divide) in MODULE 2.

Now for these changes from table.5 it can be seen that function 1(i.e. main) in MODULE 1 has been detected by faults F2 and F3 and so binary value is 1 in position 2 and 3 in Fault_module[ ].

For function 2(i.e. factorial) in MODULE 1 has been detected by faults F4 and so binary value is 1 in position 4 in Fault_module[ ].

For function 2(i.e. divide) in MODULE 2 has been detected by faults F5 and so binary value is 1 in position 5 in Fault_module[ ].

F1	F2	F3	F4	F5
0	1	1	1	1

Similarly Weight_factor array can be calculated from the above Fault-module table and Weight factor computational algorithm as

Example: For fault1 from fault-module table in the case study the weight factor is

$$\frac{\text{line\_where\_fault\_inroduced}}{\text{fuction\_length}} = \frac{3}{5} = 0.6$$

F1	F2	F3	F4	F5
0.6	0.63	0.2	0.58	1

Now we have both Fault_module[ ] and Weight_factor[ ] .

Therefore in proposed approach we calculate the nearest neighbour of Fault_module array with respect to each row Tj in Fault-testcase table.3.
According to proposed algorithm the calculation is made only if it is 1 in Fault_module[] array otherwise we neglect that term in the array.
In this example indexes under consideration are 2, 3, 4 and 5 from Fault_module array and the first index is not considered as it is 0.

For test case 1 i.e T1

F1	F2	F3	F4	F5
1	0	0	1	1

By algorithm the distance is $\{(1-0)*0.63+ (1-0)*0.2+ (1-1)*0.58+ (1-1)*1\}^{1/2} = 0.911043$

For test case 2 i.e. T2

F1	F2	F3	F4	F5
0	1	0	1	1

By algorithm the distance is $\{(1-1)*0.63+ (1-0)*0.2+ (1-1)*0.58+ (1-1)*1\}^{1/2}$
$= 0.4472135$

For test case 3 i.e. T3

F1	F2	F3	F4	F5
0	0	1	0	1

By algorithm the distance is $\{(1-0)*0.63+ (1-1)*0.2+ (1-0)*0.58+ (1-1)*1\}^{1/2}$
$=1.1$

For test case 4 i.e. T4

F1	F2	F3	F4	F5
0	0	0	1	0

By algorithm the distance is $\{(1-0)*0.63+ (1-0)*0.2+ (1-1)*0.58+ (1-0)*1\}^{1/2} =1.352774$
For test case 5 i.e. T5

F1	F2	F3	F4	F5
0	1	0	0	1

By algorithm the distance is $\{(1-1)*0.63+ (1-0)*0.2+ (1-0)*0.58+ (1-1)*1\}^{1/2}$
$= 0.883176$
Therefore by nearest neighbour concept the test case with least distance should be checked or tested first so it has high priority and the rest follow the same procedure.

Therefore the prioritized test cases are:
T2, T5, T1, T3, T4

8. ANALYSIS OF EXPERIMENTAL RESULTS

It can be seen that the new prioritized test cases are T2, T5, T1, T3, T4 from the above results.

Which can be easily verified that test case 2(T2) covers fault 2 fault 4 and fault 5 from Table .6 of section 6 i.e. fault-test case. Which are the corresponding faults of the function which have been modified i.e. main( ), factorial( ) and divide( ). So it proves that the test case 2 is the most prioritized one as it is handling all the faults generated by those functions.
Similarly test case 5(T5) covers fault2 and fault5 from Table .6 of section 6 i.e. fault-testcase which covers two functions i.e. main( ) and divide( ) of the three functions.so it implies T5 is the second most prioritized test case.
Similarly the priority of the other test cases is validated.

9. QUANTIFYING TESTABILITY

Quantification is not an easy task in the software development process and effort are made to formulate the process using RIP and Mutation concept in the software. Mathematical formulations are possible for testability [23, 24, and 25] and faults can be induced as shown in fig.2.

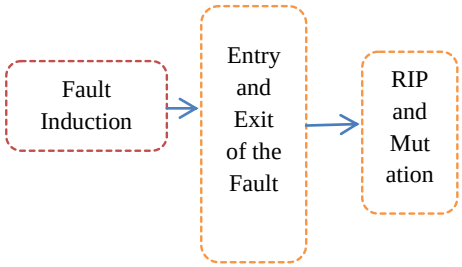

Figure *2*

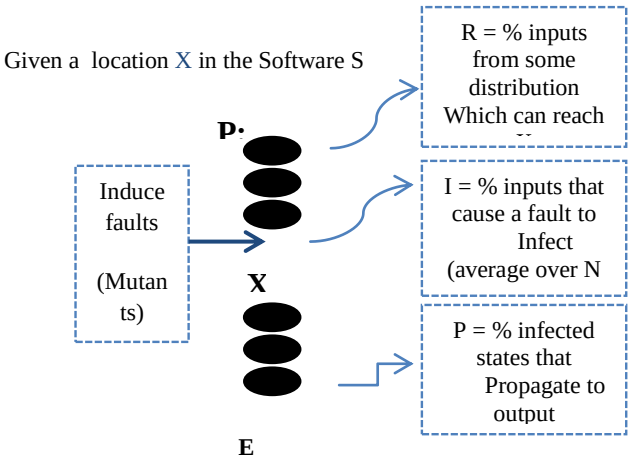

Figure3

Sensitivity (X) = R * I * P

Testability (P) = F (Sensitivity (X)), for all X in P

Sensitivity and Testability are formulated based on the fault induction mechanism which is shown in figure3.

10. CONCLUSION

In this paper a new approach for prioritizing the test cases that takes in to account the test cases and its faults, modules and its respective faults is presented. This new technique using data mining is flawless according to the experimental results and this new approach is promising in terms of ordering the test cases in test suites so as to detect faults early in the testing process.

11. ACKNOWLEDGEMENT

The author would like to thank the research department of Sohar University and Faculty of Computing and IT, Sohar University, Oman and Faculty of Engineering and IT, University of Queensland, Brisbane, Australia for the research support.

12. REFERENCES

[1] Dinesh Kumar Saini and Nirmal Gupta "Class Level Test Case Generation in Object Oriented Software Testing, International Journal of Information Technology and Web Engineering, (IJITWE) Vol. 3, Issue 2, pp. 19-26 pages, march 2008. USA

[2] Dinesh Kumar Saini "Testing Polymorphism in Object Oriented Systems for improving software Quality" ACM SIGSOFT Volume 34 Number 2 March 2009, ISSN: 0163-5948, USA

[3] S. Elbaum, A. G. Malishvesky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. on Software Engineering*, 28(2):159–182, February 2002.

[4] Dinesh Kumar Saini and Moinuddin Ahmad, "Return on Investment and Effort Expenditure in the Software Development Environment", International Journal of Applied Information Systems 4(7):35-41, December 2012. Published by Foundation of Computer Science, New York, USA. BibTeX 10.5120/ijais12-450813

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. onSoftware Engineering*, 27(10):929–948, Oct. 2001

[6] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, Alexey G. Malishevsky, Selecting a Cost-Effective Test Case Prioritization Technique, April 20, 2004.

[7] Lakshmi Sunil Prakash, Dinesh Kumar Saini and Kutti N.S. "Integrating EduLearn Learning Content Management System (LCMS) with Cooperating Learning Object Repositories (LORs) in a Peer to Peer (P2P) architectural Framework" ACM SIGSOFT Volume 34 Number 3 May 2009, ISSN: 0163-5948, USA.

[8] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel, Sebastian Elbaum, CostcognizantTest Case Prioritization, 2006

[9] Elbaum, S., Malishevsky, A., and Rothermel, G. Incorporating varying test costs and fault severities into test case prioritization, IEEE Computer Society, Washington, DC, 329-338.2001.

[10] Introductio To Data Mining by Pang-Ning Tan, Michael Steinbach , Vipin Kumar Belur V. Dasarathy, editor (1991) Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques, ISBN 0-8186-8930-7

[11] Dinesh Kumar Saini and Nirmal Gupta "Fault Detection Effectiveness in GUI Components of Java Environment through Smoke Test", Journal of Information Technology, ISSN 0973-2896 Vol.3, issue3, 7-17 September 2007.

[12] Horgan, J.R and S. London, ATAC: A data flow coverage testing tool for C, May 1992,pp 2-10, IEEE press.

[13] Dinesh Kumar Saini "Software Testing for Embedded Systems" International Journal of Computer Applications , Published by Foundation of Computer Science, New York, USA, Volume 43 Number 17, April 2012 Page No. 1-6. 10.5120/6192-8700.

[14] Aggarwal K. K., Y. Singh and A. Kaur (2004),"Code Coverage based technique for prioritizing test cases for regression testing", SIGSOFT Software engineeringNotes, 29(5):1-4.

[15] Dinesh Kumar Saini and Moinuddin Ahmad, "Business Aspect of Software Reusability", International Journal of Applied Information Systems 4(7):28-34, December 2012. Published by Foundation of Computer Science, New York, USA. BibTeX, 10.5120/ijais12-450812

[16] Kim J. and A. Porter, A history based test prioritization technique for regression testing in resource constrained environments, ICSE 2002, pp 119-129,New York.

[17] Srivastava A. and J. Thiagarajan, Effectively prioritizing test cases in development environment, ISSTA, pp 123-133, july-2002, New York.

[18] Dinesh Kumar Saini, Lingaraj A. Hadimani and Nirmal Gupta "Software Testing Approach for Detection and Correction of Design Defects in Object Oriented Software" Journal of Computing, Volume 3, Issue 4, April 2011, ISSN 2151-9617, Page No. 44-50

[19] Srikanth H. and L. Williams, On the economics of requirements based test case prioritization, EDSER 2005, pp 1-3, New York.

[20] Do, H. G. Rothermel and A. Kineer, "Empirical studies of test case prioritization in a JUnit testing environment, 2004, pp 113-124, Los Alamitos, IEEE press.

[21] Do, H. G. Rothermel and A. Kineer, Prioritizing JUnit Test cases : An empirical assessment and cost benefits analysis, Empirical Software engineering, March 2006.

[22] Dinesh Kumar Saini and Lakshmi Sunil Prakash, "Plagiarism Detection in Web based Learning Management Systems and Intellectual Property Rights in the Academic Environment", International Journal of Computer Applications 57(14):6-11, November 2012. Published by Foundation of Computer Science, New York, USA. BibTeX 10.5120/9180-3598.

[23] Dinesh Kumar Saini "A Mathematical Model for the Effect of Malicious Object on Computer Network Immune System" Applied Mathematical Modelling, 35(2011) Page No. 3777-3787 USA, doi:10.1016/.2011.02.025.

[24] Dinesh Kumar Saini "Security Concerns of Object Oriented Software Architectures" International Journal of Computer Applications, Feb 2012, 40 (11), Page No. 41-48.

[25] Dinesh Kumar Saini and Yashvardhan Sharma "Soft Computing Particle Swarm Optimization based Approach for Class Responsibility Assignment Problem" International Journal of Computer Applications, Feb 2012, 40 (12), Page No 19-24.

[26] Hemraj Saini and Dinesh Kumar Saini "Malicious Object dynamics in the presence of Anti Malicious Software" European Journal of Scientific Research ISSN 1450-216X Vol.18 No.3 (2007), pp.491-499 © Euro Journals Publishing, Inc. 2007 http://www.eurojournals.com/ejsr.htm EUROPE