Role of Quality Source Code Documentation in Software Testing

Prem Parashar[(1),] Arvind Kalia[(2),], and Rajesh Bhatia[(3)]

(1) Faculty of Applied Science and Technology .Sheridan College (Canada)
E-mail: prem.parashar@gmail.com

(2) Department of Computer Science.Himachal Pradesh University (India)
E-mail: arvkalia@gmail.com

(3) Department of Computer Science.PEC University of Technology (India)
E-mail: rbhatiapatiala@gmail.com

ABSTRACT

Software testing is performed to validate that software under test meets all requirements. With the increase in software developing platforms, developers may commit those errors, which, if not tested with appropriate test cases, may lead to false confidence in software testing. In this paper, we proposed that building quality source code documentation can help in predicting such errors. To validate this proposal, we performed an initial study and found that if software is well documented, a tester may predict the possible set of errors that developers may commit, and hence, may select better test cases that target those faults. From this study, it has been observed that proper code documentation can help in selecting appropriate test cases from candidate test cases and can lead to more effective software testing.

Keywords: Software testing, Documentation, Prioritization, test suite, faults.

1- INTRODUCTION

Software testing is an important software development activity. The objective of software testing is not limited to validate that the software is doing what it is expected to do but to ensuring that software is not doing what is not expected. Software testing often consumes 50-60% of the total development time and cost [1]-[5].

Software documentation plays a significant role in software design, development, testing, and implementation. If documentation exists, this is the first document that its users refer to understand the intended behavior of software [8]. Different software documents are referred to by different classes of user. For example, code documentation is referred to by the core development team, test documents are used by the test team, and help and installation manuals are generally used by end users. The software documentation is written internally in the form of software comments or externally in the form of software design documents, help manuals, UML diagrams, installation manuals and other related materials [1], [7]-[11]. Though documentation is not usually deemed as important as software itself, but with the evolution of soft-

ware development techniques, software documentation has been taken as benchmark for understanding, development, implementation and post-implementation [4], [12]-[14].Software are evolved more frequently these days; hence, documentation must be updated when software changes.

Software documentation helps in understanding the behavior of software. Building quality source code may assist the testing team to build test cases that are more effective for the system under test. Though the use of documentation can be extended to any phase, we highlighted its use in the area of software testing. The current study emphasized the use of detailed and complete source code documentation in effective test case selection from a set of candidate test cases. By candidate test cases, we mean the test cases which have identical code coverage and are intended to expose the same faults. For example, to check whether an integer is even, the numbers 2, 4, 6, 10 etc. are the candidate test cases and hence, any test case may be selected for execution. By detailed and complete documentation, we mean that every major part of software is well documented. Let us assume that our application involves sorting (in ascending order) a list of integers. There are a number of methods to accomplish this task. Therefore, software documentation should contain the details of the method (bubble, selection, quick, etc.) that has been chosen for the current application.

This paper is organized as follows. Section 2 discusses about the background work in this field. Section 3 contains the research methodology, section 4 describes the specific objectives, section 5 talks about experimental setup and results, and section 6 concludes the paper with future scopes.

2- RELATED WORK

Software testing is very expensive and crucial phase of software development due to change in the requirements, high software dependency, and changing software development paradigms. Software documentation plays significant role in testing. Forward and Lethbridge [7],[15],[16] conducted a detailed survey to find out the relevance of software documentation in effective software development, maintenance and future enhancement. The survey was conducted by asking software developers about the software documentation they refer to while developing a project. There was no any clear indication that the software documentation depends on the type of software development model, however, the participants agreed that agile and iterative software development involve more up-to-date software documentation as compared to water-fall software development. The different studies also revealed that although up-to-date software documentation is always more effective and desired, outdated software documentation may also help in understanding software up to some extent. As software evolves, so should the software documentation [17]. In real practice, it has been observed that during the maintenance of software, software documentation is generally not updated due to strict time and other cost constraints [18].

Hartmann et al. [6] found that software documentation is very effective for understanding and implementing complex systems. Their study further revealed that proper software documentation is especially important in reverse engineering, understanding legacy software, and Mining Software Repositories (MSR). By proper software documentation, they meant that documentation is not only limited to software coding but it covers all aspects of software.

Source code documentation has been found very useful in the software maintenance. The results of a survey conducted by Sergio et al. [13] revealed that after source code, comments are the form of software documentation that is most frequently referred to when building structured analysis and object-oriented artifacts. Contrary to general opinion, it was observed that the UML diagrams (except class and use case diagrams) are referred only at the time of software development, and rarely referred after the implementation of software. Their survey also revealed that quality software documentation is either absent in legacy software or it is not up-to-date. The documentation generally not updated after the maintenance of software.

It has been observed that internal software documentation, which is accomplished by writing code comments, is generally not updated as the software evolves [19]. Comments that are useful at one time of software development may become irrelevant when they are not updated as the software changes. Bad comments may mislead software developers due to obsolete and irrelevant information. In a study conducted by Lin T. et al. [19], it was found that generally comments are not maintained when software is modified. The obsolete comments may lead to new bugs at the time of maintenance. The main barrier for not maintaining the comments was the language in which they are written. The comment writing is performed in natural languages which are highly imprecise and hence it's difficult to automate documentation. The automatic generation of user documentation for GUI programs for information systems was proposed by Tsybin, A. and Lyadova, L. [20]. They proposed automatic user documentation which is based on the metadata maintained in documented systems. The document component is treated as a spate entity which can be integrated with any other entities to derived meaningful results and up-to-date enhancement in it.

In an interview study, performed by Timea and Barbara [21], it has been observed that a lot of information is needed by testers at the time of test case formation and software testing. Up-to-date information certainly helps the testing team to make right decisions about testing process. The study is confined to conclude which of the available documentation records are most frequently used in software testing. The experienced testers of SIKOSA project were the subjects of interview. The results of this study showed that previous defect reports, if exist, are the most frequently used documents during system testing. User manual and requirement documents are also referred as they provide the information about the expected behavior of system under test. Though, such studies seem very beneficial for future development and testing,

their universal acceptance requires more studies that provide broader view on the subject topic.

The software documentation is very useful in understanding, developing, testing, and maintaining software. The incorrect or obsolete documentation, especially user manual documents, can put the prestige of a software developing company into risk. The software is expected to behave as per user manual, and if not, software developing company may subject to the liability for breach of promises [22]. It might seems self evident that the main reasons of not updating software documentation with the software evolution is that software documentation has still been considered a pure non-technical attribute of software which does not participate in its direct implementation; hence, it has been considered by software developing houses as an extravagant. Software companies hesitate to allot sufficient budget and man power for software documentation activities especially at the time of its maintenance and most of the software developing firms consider it as a spare/extra time activity [7, 16].

3- RESEARCH METHODOLOGY

The types of error that may occur during the development of software depend upon many factors such as typographical errors, mixed-syntax errors(two different programming languages having same syntax with different meaning), the programming approaches used, the skill level of software developers, the size of the module, and types of software (application, system , embedded etc.). In this paper, we emphasized on the typographical and mixed- syntax errors committed by developers that are not reported by compilers. For the motivation of this study, we considered an example of writing language C code for swapping two integers A and B. Depending upon the types of developers, the following set of codes may be used (with third variable, without using third variable, using bit-wise operator):

(a)	(b)	(c)
Temp = A;	A=A+B;	A=A^B;
A = B;	B=A-B;	B=A^B;
B = Temp	A=A-B;	A=A^B;

Blocks (a), (b), and (c) may contain different types of typographical bugs and may require different test cases to detect them. Though, we have highlighted only three ways, there may be hundreds other ways to accomplish this task. Hence, documenting the method selected from a set of candidate methods will certainly help the testing team to get some clue about the possible typographical errors as well as the test cases that may not be relevant to the code.

For example, if the code block (a) is selected for current application, a tester will be getting better idea about the kind of errors that may occur in it by running a test case. Even, a test case, that looks very promising, if executes the erroneous block and does not trace out the intended errors, should be discarded from the final test suite. We considered three sets (a1, a2, a3) of buggy code for block (a).

(a1)	(a2)	(a3)
A = Temp;	Temp=A;	Temp = A;
A = B;	B =A;	B = Temp;
B = Temp	A = Temp	A = B;

From the outcome of the test cases we may conclude which block of code has been executed. The outcomes of three blocks (a1, a2, a3) for two test cases have been shown in Table 1. From this table, it is clear that if two integers are equal, only block (a1) gives error, where as other two do not, but if A and B are different all of the three blocks give erroneous output. By looking into the pattern of outcomes, a tester may conclude about the possible errors and can send the useful feedback to the developing team. We assume that random value assigned here will be different from the input values, although that case may occur any time and can fail a test case.

Table 1 Test Cases with outcomes
R-Random value, P-Pass, F-Fail

TestCase	**OutCome**		
TC(A,B)	**a1(A,B)**	**a2(A,B)**	**a3(A,B)**
TC(4,2)	(2,R)**P**	(4,4)**P**	(4,4)**P**
TC(4,4)	(4,R)**P**	(4,4)**F**	(4,4)**F**

4- MAIN OBJECTIVE

The main objective of this study is to find how software code documentation can facilitate software testers to estimate about the possible buggy code. The particular questions of the current research are:

- Can software documentation be helpful in optimum test case selection from the set of candidate test cases?
- Can software documentation provide any clue to software tester about the typographical errors that may occur during software development?

```
1.  #include <stdio.h>
2.  void reorder(float*a,float*b, float*c)
3.  {
4.  float temp1, temp2;
5.  if((*b>*a)&&(*a>*c))→F4
6.  {
7.  temp2=*a; *a=*b; *b=temp2;
8.  }
9.  else
10. if((*a>*c)&&(*c>*b))→F7
11. {
12. temp1=*b; *b=*c; *c=temp1;
13. }
14. else
15. if((*c>*a)&&(*a>*b))→F8
16. {
17. temp1=*b; temp2=*a; *a=*c; *b=temp2;
18. *c=temp1;
19. }
20. else
21.  if((*b>*c)&&(*c>*a))//  →F9
22. {
23. temp1=*c; temp2=*a; *a=*b;
24. *b=temp1; *c=temp2;
25. }
26. else
27. if((*c>*b)&&(*b>*a))→F10
28. {
29. temp2=*a;
30. *a=*c;
31. *c=temp2;
32. }
33. }
34. int main()
35. {
36. char type;
37. float a,b,c,s,area
38. cout<<"Enter sides:(a,b,c)\n";
39. cin>>a>>b>>c;
40. reorder(&a,&b,&c);
41. if(a<=0 ||(b<=0 ) ||(c<=0))
42. {
43. cout<<"Wrong Input\n\n";
44. goto label;
45. }
46. else
47. if(b+c<=b)//if(b+c<=a)→F1
48. {
49. cout<<"Invalid\n";
50. goto label;
51. }
52. else
53. if((a==b)&&(b==c))
54. {
55. type='e';
56. cout<<"Equilateral \n";
57. }
58. else
59. if((a==b)||(a==c)) →F2
60. {
61. type='i';
62. cout<<"Isosceles:\n";
63. }
64. else
65. if(a*2==(b*2+c*2)) →F3
66. {
67. type='r';
68. cout<<"Right Angled :\n";
69. }
70. else
71. if(a!=b!=c)//if((a!=b)&&(b!=c)))→F11
72. {
73. type='s';
74. cout<<"Scalene Triangle:\n";
75. }
76. else
77. {
78. cout<<"No-Category\n"; goto label;
79. }
80. switch(type)
81. {
82. case 'r':
83. area=b*c/2;
84. break;
85. case 'e':
86. area=a*2*sqrt(3)/4; →F5
87. break;
88. case 'i':
89. case 's':
90. s=(a+b+c)/3;//floats=(a+b+c)/2;→F6
91. area=sqrt(s*(s-a)*(s-b)*(s-c));
92. }//end switch
93. cout<<"Area of the Triangle:"<<area;
94. label: return 0; } //end main
```

Figure 1 TriTypArea program

5- EXPERIMENTAL SETUP AND RESULTS

5-1 Experimental Setup

In order to check the results of our proposal, we have coded a program ***TriTypArea*** (Fig. 1) in language C which is a modified version of a very popular program ***TriTyp.java*** (http://cs.gmu.edu /~pammann /637/ code/ triangle/TriTyp.java). This program determines the type of a triangle (Scalene, Equilateral, Isosceles, Right Angle, and Invalid triangle) on the basis of its dimensions. The program ***TriTyp*** has been considered by many researchers for software testing [3],[23],[24] due to its simplicity and effectiveness. In order to determine the time of execution of test cases, we have taken following functions:

QueryPerformanceFrequency();

QueryPerformanceCounter();

To find the accuracy of test case execution time, we executed a test case thrice and then calculated the average time of its execution. The program ***TriTypArea*** has been executed using Dev-C++ 4.9.9.2 compiler and computer with specifications, Intel® Core™ 2 Duo Core CPU T9300 @2.50 GHZ and 1.99 GB RAM. ***TriTypArea*** compared to ***TriTyp,*** performs two additional functions 1) arrange the sides in descending order, and 2) calculate the area of triangle. Thus, our candidate program not only determines the type of a triangle but also calculates the area of a valid triangle.

We have not imposed any constraints [5],[23],[24] on entering sides of triangle as it made our program more user friendly and arranging the sides in descending order helped in reducing the size of program. We have used seeded faults (shown in Fig. 1 with the help of bold lines), which are very close to actual coding scenario. The partial set of test cases, seeded faults and the time of execution of test cases have been shown in Table 4. The grey cells in the table represent the disagreement between expected and actual outcome, and hence represent a successful test case. The black cells represent an unsuccessful test case, i.e. one which does not reveals the intended faults. The formation of this partial test suite has been influenced by [9] and Boundary Value Analysis (BVA).

5-2 Results and Analysis

The area of a triangle can be calculated either by using specific formula corresponding to the type of triangle or general Heron's Formula. The different formulae are coded differently and hence may lead to different types of bugs. To detect these bugs, specific set of test cases is needed. We have identified

some expressions (*Exp1, Exp2, Exp3*), which generate identical outcome for same input value. We believe that there may be numerous such expressions in real world that may mislead software testing team due to their magical characteristics. Hence, from testing perspectives, the set of inputs that satisfies these expressions, must be excluded. e.g.

$$a+a = a*a \quad ¥\ a\{0,2\} \qquad (Exp1)$$
$$a^p = a^q \quad ¥\ a\{0,1\} \qquad (Exp2)$$
$$a^2 = a*2 \quad ¥\ a\{0,2\} \qquad (Exp3)$$

The presence of such test cases in the final test suite may be alarming as their execution does not expose the faults and provides fake confidence in the software testing. The situation may arise specifically in regression testing where a tester has limited time and may be one test case is selected for execution from each category of candidate test cases.

If the software documentation provides any clue of the presence of expressions which generate same value for some input, a tester can easily exclude those test inputs. Such practice not only helps in building confidence in software testing but also avoids the execution of a fail test case. By fail test case, we mean a test case that does not expose the bug it is intended to expose.

In the test suite as shown in Table 3, test case TC7 (2, 2, 2) is such a test case. It does not expose fault F5 due to the reason that fault F5 i.e. **area=a*2*sqrt(3)/4** contains *Exp3* in it. We have discussed in this section that *Exp3* should not be executed for the input value a{0,2} due to its misleading outcome. We consider such test cases very dangerous to the application and we argue that such test case execution should be avoided at the time of testing.

In this study, it has been observed that the fault F11 (Fig. 1: shown with dark black line) exists in the code but there is no test case that surfaces it out. The set of test cases that may surface out this fault require a>b>(c=1). While analyzing the above expression, it was found that the test case that may cover F11 is actually a test case that represents invalid triangle. Since, the invalid triangle is top in the hierarchy of nested-if structure, therefore the control will never come to condition corresponding to fault F11.This fault will only be exposed if the order of the conditions is changed and F11 condition is prioritized in the nested-if structure. This rearrangement may further lead to some semantic errors. It is believed that software testing is an art and the outcome of software testing not only depends on the test suite execution but its success depends on many other factors like tester state-of-mind at the time of execution, time constraint, time-of–day of test suite execution, and tester's experience etc. [9],[19],[25]. In the current practice of software development, software testing goes parallel with software development, therefore, software testing team finds time for deep analysis of code behavior and hence, they can form better test suite. For the final execution of test cases, at the time of regression testing, we have applied fault-based prioritization method presented

in [26] which select a test case which determines a fault that has been allotted minimum time for its execution in the given test suite. The main steps of the technique have been highlighted in Fig. 2:

```
   Inputs:
       TF: Total faults not yet detected
       t: permissible time budget
       Fi: ith. fault
       T:set of test cases
Ouput:
       P:prioritized test suite
Prioritize( T,P,TF, t)
   1. repeat (step 2- step 6) until TF is empty or t >TB
   2. Select fault Fi from TF which takes least time for execution.
   3. If Fi is detected by more than one test case say Ti and Tj, select the one which detects it with minimum time. Let the test case selected be Ti..
   4. Include test case Ti to P.
   5. Minus fault Fi and other faults detected by test case Ti from TF. Add time ti to t.
   6. Minus Ti from T.
```

Figure 2 Main steps of the prioritization algorithm

Table 2 Prioritized Test Suite for Regression Testing

Test Case	Faults- Exposed	Time(μs)
TC08	F5	223
TC09	F2,F6,F9,F10	169
TC10	F6,F7,F8	127
TC11	F4,F6	103
TC35	F1,F3	333

The prioritized set as a result formed has been shown in Table 2. For the detection of F5, the test case TC08 has been selected from the candidate test cases, though test case TC07 seems more promising due to its time of execution. If software documentation is not in place, TC07 would have been selected for final execution and it would not have exposed the intended fault. Thus, software documentation not only helps in appropriate test case selection but also provide meaningful feedback to developing team. The developing team will find it easy to fix the bugs as they are provided with the information about the particular code that is erroneous with the possible type of errors.

Table 3 Partial Test suite for TriTypArea (*gray cells show faults*)

TC	a,b,c	Class	Act_Class	Area	Act_Area	Fault	(µs)
TC1	1,2,3	Invalid	Right Anled	NA	1.0	F1	293
TC2	1,3,2	Invalid	RightAngled	NA	1.0	F1	335
TC3	2,1,3	Invalid	RightAngled	NA	1.0	F1	385
TC4	2,3,1	Invalid	RightAngled	NA	1.0	F1	406
TC5	3,1,2	Invalid	RightAngled	NA	1.0	F1	423
TC6	3,2,1	Invalid	RightAngled	NA	1.0	F1	448
TC7	2,2,2	Equi	Equi	1.73	1.73	NIL (Exp. F5)	208
TC8	3,3,3	Equi	Equi	3.89	2.59	F5	223
TC9	3,4,4	Equi	Scale	5.56	0.52	F2,F6,F9,F10	169
TC10	4,3,4	Equi	Iso	5.56	0.52	F6,F7,F8	127
TC11	4,4,3	Equi	Iso	5.56	0.52	F4,F6	103
TC12	3,4,5	Right	Scale	6.0	0.0	F3,F6	279
TC13	3,5,4	Right	Scale	6.0	0.0	F3,F6	241
TC14	4,3,5	Right	Scale	6.0	0.0	F3,F6	215
TC15	4,5,3	Right	Scale	6.0	0.0	F3,F6	225
TC16	5,3,4	Right	Scale	6.0	0.0	F3,F6	250
TC17	5,4,3	Right	Scale	6.0	0.0	F3,F6	270
TC18	4,5,6	Scalene	Scale	9.92	0.0	F6	162
TC19	4,6,5	Scalene	Scale	9.92	0.0	F6	217
TC20	5,4,6	Scalene	Scale	9.92	0.0	F6	243
TC21	5,6,4	Scalene	Scale	9.92	0.0	F6	276
TC22	6,4,5	Scalene	Scale	9.92	0.0	F6	308
TC23	6,5,4	Scalene	Scale	9.92	0.0	F6	331
TC24	1,1,100	Invalid	Iso	NA	1.0	F1,F6,F8	127
TC25	1,100,1	Invalid	Iso	NA	1.0	F1,F6,F9	218
TC26	100,1,1	Invalid	NO-category	NA	NA	F1	127
TC27	1,2,200	Invalid	NO-category	NA	NA	F1	443
TC28	1,200,2	Invalid	NO-category	NA	NA	F1	368
TC29	2,200,1	Invalid	NO-category	NA	NA	F1	323
TC30	2,1,200	Invalid	NO-category	NA	NA	F1	281
TC31	200,1,2	Invalid	NO-category	NA	NA	F1	241
TC32	200,2,1	Invalid	NO-category	NA	NA	F1	201
TC33	1,199,200	Invalid	Right	NA	99.5	F1,F3	355
TC34	1,200,199	Invalid	Right	NA	99.5	F1,F3	367
TC35	199,1,200	Invalid	Right	NA	99.5	F1,F3	333
TC36	199,200,1	Invalid	Right	NA	99.5	F1,F3	380
TC37	200,1,199	Invalid	Right	NA	99.5	F1,F3	382
TC38	200,199,1	Invalid	Right	NA	99.5	F1,F3	380
TC39	100,199,1	Invalid	NO-category	NA	NA	F1,F3	452
TC40	199,1,100	Invalid	NO-category	NA	NA	F1,F3	494
TC41	199,100,1	Invalid	NO-category	NA	NA	F1,F3	474
TC42	1,100,100	Iso	Scale	49.99	2194.44	F1,F6,F9, F10	466
TC43	100,1,100	Iso	Iso	49.99	2194.44	F7,F8,F6	457

5-3 Threat to validity

The main threat to the validity of this study is that the selection of appropriate test cases for the testing is done manually.Such selection may be errorenous and time cosuming especially for large or complex software. The results of this study are mainly depending upon the internal software code documentation.The study can not be implemented to software which have not been documented properly or where the software documentation is not updated with the software modification.

6- CONCLUSION

In this paper, we highlighted the importance of software documentation in the field of software testing. From the experimental setup, it is clear that if proper software documentation is in-place, it can help the testing team in building and executing relevant test cases. The outcome of such testing strategies will be very useful to software industry. The study also highlighted the faults which are present but are not detected by the present set of test cases due to their unreachable positions in the software. To the best of our knowledge, software documentation so far has been used to understand the behavior of software for better software development and its maintenance but it has not been related to software testing for test suite minimization, reduction, or selection. Nowadays, the testing has become very challenging due to software evolutions, different software development approaches, tight time constraints, and software complexity. Thus, it is highly desirable to form a test suite that detects maximum faults or performes maximum code coverage with in allotted time. Though, in this study, it has been observed that the volume of fail test cases is very low, but we believe that the execution of such test cases is very alarming and misleading to testing team as it hides the faults and provide false confidence in software testing. Thus, their detection and elimination from the final test suite is highly desirable. The detection of such test cases may be done manually for small software systems but for large and complex systems, this process need to be automated. In the current study, we have used only arithmetic expressions which may produce same output for some set of inputs; the study can be extended further for all types of expressions. This prioritization technique based on the software documentation can be very beneficial for successful testing of big software which may contain thousands of methods and each method in terms may have several canditate methods.

REFERENCES

[1] B. Beizer, Software Testing Techniques, Second Edition. Published by dreamtech, New Delhi, pp. 135-143, 2008.

[2] E. Engstrom, M. Skoglund, and P. Runeson, “Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review,” Proc. of ESEM’08, Oct 9-10, pp. 22-31, 2008.

[3] A. Gotlieb, M. Petit. “Path-Oriented Random Testing,” Proc. IWRT, pp. 28-35, 2006.

[4] P. Parashar, A. Kalia, R. Bhatia, “Pair-wise Time-ware Test Case Prioritization for Regressiojn Testing,” Proc. ICISTM 2012, CCIS 285, pp. 176-186, 2012.

[5] S. Yoo, and M. Harman, “Regression Testing Minimization, Selection and Prioritization: A Survey,” Soft. Test. Verif. Reliab., pp. 67-120, 2007.

[6] J. Hartmann, S. Huang, S. Tilley, “Documenting Software Systew with views II: An Integrated Aproach Based on XML,” SIGDOC’01, pp. 237-246, 2001.

[7] A. Forward, and T.C. Lethbridge, “Software Engineering Documentation Priorities: An Industrial Survey,” Submitted to CASCON, pp. 23-31, 2002.

[8] W. John. “Practical Support for CMMI-SW-Project Documentation using IEEE Software Engineering Standards,” C-SPIN’06, pp. 267-280, 2006.

[9] G.J. Mayers, The Art of Software Testing, Second Edition.Published by John Wiley and Sons Inc. Hoboken, New Jersey, pp. 107-121, 2004.

[10] R.S. Pressman, Software Engineering :A Practitioner’s Approach, Fifth Edition. Publisher Thomas Casson, pp. 450-467, 2001.

[11] M. Ratzmann, and C.D. Young, Software Testing and Internationalizaion. Lemoine International, Inc. Salt Lake City, pp. 241-247, 2003.

[12] P. Parashar, A. Kalia, R. Bhatia, “Change Impact Analysis: A Tool for Effective Regression Testing,” Proc. ICISTM 2011, CCIS 141, pp. 160-169, 2011.

[13] C.B. Sergio, A. Nicolas , and M.O. Kathia, “A Study of the Documentation Essential to Software Maintenance,” In SIGDOC’05, pp. 68-75, 2005.

[14] T. Xie, N. Tillmann, D.H. Jonathan, “Future of Developer Testing : Building Quality in Code,” Proc. of FoSER’2010, pp. 415-420, 2010.

[15] A. Forward, and T.C. Lethbridge, “The Relevance of Software Documentation, Tools and Technologies: A Survey,” Proc. Of DocEng’02, ACM, New York, pp. 26-33, 2002.

[16] A. Forward, and T.C. Lethbridge, “Qualities of Relevant Software Documentation: An Industrial Study,” Proc. ICSE 2003, pp. 63-71, 2003.

[17] B. Thomas, and S. Tilley, “Documentation for Software Engineers: What is needed to aid system understanding,” SIGODC’01, pp. 235-236, 2001.

[18] V. Marcello, and R.C. Curtis, “An overview of Industrial Software Documentation Practices,” CONICYT, pp. 179-186, 2000.

[19] L. Tan, D. Yuan, and Y. Zhou, “HotComments: How to Make Program Comments More Useful?,” HotOS’07, pp. 49-54, 2007.

[20] A. Tsybin, and L. Lyadova, “Software Testing and Documenting Automation,” International Journal of Information Technologies and Knowledge”. Vol. 2, pp. 267-272, 2008.

[21] I. Timea, and P. Barbara, “On the Role of Communication, Documentation and Experience during System Testing: An Interview Study,” Proc. PREMIMUM’08, pp. 23-47, 2008.

[22] C. Kaner, “Liability for Defective Documentation,” Proc. SIGDOC’03, pp. 192-197, 2003.

[23] D. Berndt, L. Fisher, J. Johnson, J. Pinglikar, A. Watkins, “Breeding Software Test Cases with Genetic Algorithms,” Proc. of ICSS’03, pp. 131-138, Jan 6-9, 2003.

[24] H. Agrawal, J.R. Horgan, E.W. Krauser, S. A. London, “Incremental Regression Testing,” Proc. ICSM 1993, pp. 348–357, 1993.

[25] E. Jon, L. Tan, L. Patrick, “Do Time of Day and Developer Experience Affect Commit Bugginess?,” Proc. MSR’11, pp. 153-162, 2011.

[26] P. Parashar, A. Kalia, R. Bhatia, “Fault-based Time-aware Test Case Prioritization for Regression Testing,” Proc. ISC’ 2011, pp. 74-83, 2011.