ISSN 1512-1232

IDENTIFYING FAULT-PRONE MODULES IN SOFTWARE FOR DIAGNOSIS AND TREATMENT USING EEPORTERS CLASSIFICATION TREE

Bassey. A. Ekanem[1], Nseabasi Essien[2]

[1]Department of Computer Science, Delta State Polytechnic, Ozoro, Delta State, Nigeria,
Email: ba_ekanem@yahoo.com
[2]Department of Computer Science, College of education, Gusau, Zamfara State, Nigeria
Email: nseabasiess@yahoo.com

Abstract

One of the major challenges facing software developers and testers is how to identify fault-prone modules in software for quick diagnosis and treatment without compromising its quality. In a bit to deliver quality and error-free software products, most testers usually spent hours on redundant and unnecessary testing due to their inability to identify modules likely to harbor harmful errors for test. Considering the amount of time, energy and resources usually lost to such, this research presents a tool termed, EEPorterS Tree capable of identifying fault-prone modules for quick diagnosis and treatment by testers. EEPorterS is christened from the surnames of the contributors, namely Ekanem, Essien, Porter and Selby. The tool is constructed based on factors likely to render a program module error-prone. Stepping modules through the tree, their ranks, which is from Rank 1 (modules likely to harbor many harmful errors) to Rank 5 (modules likely to harbor few errors) can be determined and tested in that order. EEPorterS was implemented with 20 software projects, which reveals 233(that is 73%) test hours reduction, and 64% improvement in error detection over random testing approach.

Keywords*: EEPorterS Classification Tree, Fault-prone Module, Program Errors, Test Sequence, Module Ranks, Testers.*

1.Introduction

Software organizations usually put their software products through intensive testing and debugging before deploying them for use. Testing and debugging are very important exercises during integration process and requires serious attention to ensure that the software meets the acceptance criteria specified in the software contract. Software reliability, availability and even maintenance costs depend on the quality of testing and debugging performed. To show how important this is, most software organizations spend between 50% to 80% of the total development cost on testing and debugging [1].

For a software to be completely free from error, it requires exhaustive testing and debugging. This has to do with testing and executing every instruction in the program at least once, testing every branch point in each direction at least once and ensuring that all control paths are also tested. Even with simple programs only a vanishing small part of all theoretically possible input cases can be exercised in this manner during the testing. Thus, on the basis of those input cases considered, some statements can be made as to the general behavior of all other input cases.

Since a purely statistical choice of test cases does not lead to conclusive information about the program, the concern for making the right choice of the selection of test cases becomes evident and if not properly done, the software is bound to fail the acceptance test [2]. Even where it narrowly passes the acceptance test, it may build up very high and frightening maintenance cost since the unidentified errors are bound to appear in very high frequency once the software becomes operational.

To ensure that the software is properly tested and released on schedule, some organizations normally insist on penalty-incentive contracts to penalize for milestones not achieved and compensate for those accomplished during the project. Also, some clients based on their experience on high maintenance costs during the first few months/years of acquiring a new software product always insist on warranty contracts specifying time period in which the contractor fixes all found errors without charge even when the software has passed acceptance test [3].

In view of the above, the major challenge before software developers is how error-prone modules in a software can be quickly identified for quick diagnosis and treatment to minimizing redundant and unnecessary tests to ensure timely release of the software even without compromising its integrity and reliability. This is important because redundant and unnecessary tests are "time killers" and should be avoided. However, not all defects result in failures; some stay dormant in the code and may never be noticed [4]. To address this issue, factors that may render a program fault-prone need to be identified and used to build a tool that can be used to quickly spot modules likely to harbor errors for immediate attention to minimize Integration Testing. Therefore, this paper seeks to address this challenge by considering existing tools, their strengths and weaknesses, then using such to build an improve tool that can be used by software developers and testers to identify fault-prone modules for quick diagnosis and treatment.

2. Methodology

The methodology for this research involved reviewing of existing fault detection and program testing methods, collection of program error data, and analyses of the collected data for test results.

Data were collected from 20-software projects by different groups of students. The software were tested in two ways: firstly, by using random testing approach, in which case, modules were selected randomly, tested and debugged by the students and the integration test data recorded. Secondly, by using EEPorterS tree approach – in this case, before testing began, the modules were first analyzed using the EEPorterS Classification tree to determine their ranks with respect to error availability.

After this, the modules were tested according to their ranks from rank 1 modules (i.e. those likely to harbor many errors) to rank 5 modules (i.e. those likely to harbor few errors), and the integration test data recorded, and analyzed for test results. The test results so obtained were compared with those of random testing approach to determine the efficacy of EEPorterS approach.

3. Identifying Fault-Prone Code

There are several known methods for identifying errors in software programs. The most common of these methods is testing; that is, executing the target program under different conditions in order to confirm its correct behavior. Another group of methods, program verifiers, attempt to prove mathematically that a program's behavior is correct for all possible input conditions [5]. In addition, there are static analysis tools that perform a set of limited checks without executing the program. The tools listed above are capable of reporting some possible errors in a software.

However, it is normally not feasible to test all possible program executions and it is extremely difficult to know what area of a program's execution to explore [6].

However, some techniques for identifying fault-prone code are usually based on past history of faults in similar applications. In this case, some researchers track the number of faults found in each components during development and maintenance. They also collect measurements about each component, such as module size, number of decisions, number of operators and operands, or number of modifications. Then, they generate equations to suggest the attributes of the most fault-prone components. The equation so generated are then used to suggest which of the program components should be tested first, or which should be given extra scrutiny during reviews or testing.

In most cases, basing the test process on past history does result in a large number of false error reports and numerous redundant tests. Therefore, this research explores alternative means of identifying fault-prone code based on the characteristics of the modules making up the software rather than basing it purely on past history of similar applications.

4. Porter and Selby Classification Tree Method

Classification tree can be a useful tool in detecting error-prone modules in software. The idea to use a classification tree to detect fault-prone components in a software came from Porter and Selby [7]. Porter and Selby suggest the use of classification tree analysis as a statistical technique that sorts through large arrays of measurement information, creating a decision tree to show which measurements are the best predictors of a particular attribute. In their suggestion, once a classification tree can be constructed based on factors likely to render a software error-prone, the following measurement data about each software component can be collected during software development for use in detecting fault-prone components during testing.

I) Program size in Lines Of Code (LOC)
II) Number of distinct paths through the code
III) Number of operators
IV) Depth of nesting
V) Degree of coupling and Cohesion (rated on a scale from 1 as lowest to 5 as highest)
VI) Time to code the component
VII) Number of faults found in the components already

After the measurement data have been collected, they can be analyzed using the classification tree to identify fault-prone components in the software.

To illustrate Porters and Selby suggestion, the classification tree below was used:

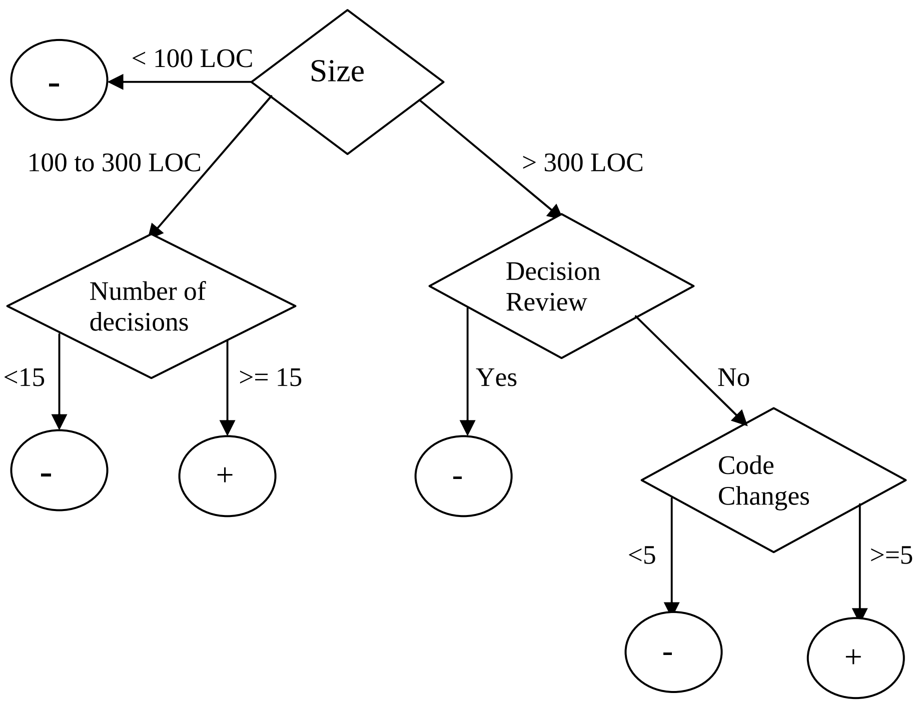


Figure 1: Classification Tree to Identify Fault-Prone Components (Source: [7])

The tree can be used by software developers to decide which components in the software are likely to have large number of faults. According to the tree, if a component has between 100 and 300 lines of code and has at least 15 decisions, then it may be fault-prone. Or if the component has over 300 lines of code, has no design review, and has been changed at least five times, then it may be fault-prone. The classification tree can help the testers in efficient testing where testing resources are limited. It can also be used to schedule inspections for such components, to help identify problems in the component before testing actually begins [8].

Though the tree can be useful in identifying fault-prone components, it is defective in the following areas:

I. It does not account for attributes like degree of coupling and cohesion, depth of nesting and number of jumps in a component.
II. It implies that any code with design review is free from error.
III. A program component having up to 15 decisions pertains poor program practice, a lower value say 5 is preferable to keep program logic as simple as possible to minimize errors.
IV. It does not cater for modern programming concepts associated with OOP and Visual programming. Object-oriented testing focuses on designing appropriate sequences of operations to exercise the states of a class [9].

5. Factors likely to make a Program Fault-Prone

For one to identify fault-prone module in software, he should be acquainted with factors likely to make a program fault-prone. These include module size, number of operand and operators, number of decisions and uncontrolled jumps, degree of coupling and cohesion, dept of nesting and frequency of path traversal. These factors were used in the construction of the EEPorterS Classification Tree.

a) Lines of Code (LOC)

Lines of code refer to the total number of program statements or lines in a program segment. The segment can be a procedure, a function, or subroutine. According to [10] small programs have error rates of 1.3% to 1.8%, with large programs increasing from 2.7% to 3.2% per line when measured against lines-of-code. A small program in this context is a program which lines-of-code are less than or equal to a hundred. Such programs are on the average are likely to have an error or none at all. The idea behind this is the longer the program the more logical it becomes hence more complex thereby increasing the chances of error occurrences. Therefore, to reduce program errors, LOC of programs modules should be kept within hundred and the logic simple enough for proper understanding.

b) Degree of Coupling and Cohesion

Modern software are modular in nature. Modularity refers to the logical partitioning of software into parts, components or simply modules. Modules are linked or connected to each other to enable them communicate effectively towards realizing the overall objective of the system. The degree of connections between modules otherwise called coupling through their module-level variables must be limited while data transfer between such modules should be minimal. Where the reverse is the case, components of such modules such as procedures, functions and subroutines are likely to harbor a lot of program errors. The higher the degree of coupling and cohesion between modules the higher the chances of being error-prone. In fact, degree of coupling and cohesion should not be beyond two modules.

c) Dept of Nesting

Loops in programs can be nested when one loop lies completely within the range of another loop. If this kind of structure is not properly handled, it can be a major source of many harmful errors in a software since control within nested loops can be tricky [11]. Therefore, EEPorterS tree considers nesting depth of 2 as suitable for accurate and reliable software. Higher depths may result in poor logic and become the major source of errors.

d) Number of Distinct Paths Through the Code

A software component that traverses many paths in the system is likely to leave errors along the path as it interacts with other components. Worse case abound where several uncontrolled jumps exist in the path. Keeping number of distinct paths not higher than 2 can allow for proper analyzes of components logic. Components going contrary to this, may harbor many errors thus deserve serious attention. This is the basis for EEPorterS tree.

e) Time to Code the Component

Duration of a software development project can adversely affect software accuracy and reliability if the time allocated to the project was not enough for thorough and careful coding. Rushing over the entire exercise to meet up with deadline is likely to result in unhealthy coding practice, which may not be without harmful errors [3].

6. EEPorterS Classification Tree

To enhance effective utilization of Porter and Selby idea, EEPorterS Classification tree is proposed.

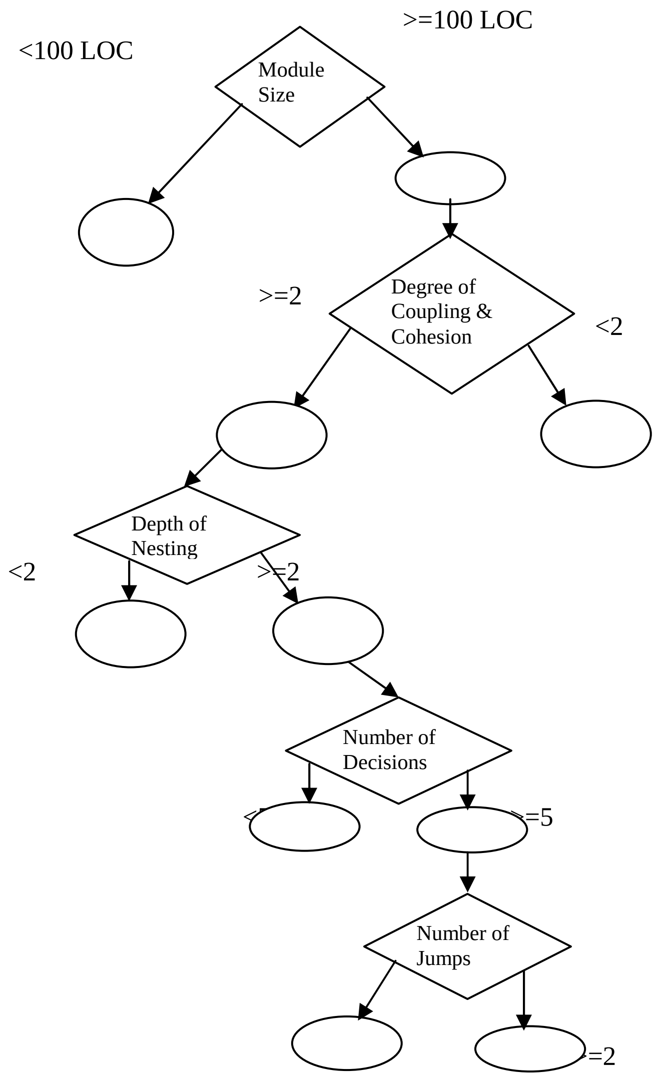


Figure 2: EEPorterS Classification Tree for Error-Prone Module Identification

EEPorterS is short for Ekanem, Essien, Porter and Selby. It is christened from the first letter of the surnames of three contributors (i.e. **E**kanem, **E**ssien and **S**elby) together with the surname of the original contributor, Porter. It is a classification tree by Ekanem and Essien based on Porter and Selby idea on using classification tree to detect fault-prone modules.

The tree is based on program attributes likely to make a program module fault-prone. These include module size, number of operand and operators, number of decisions and uncontrolled jumps, degree of coupling and cohesion, dept of nesting and frequency of path traversal.

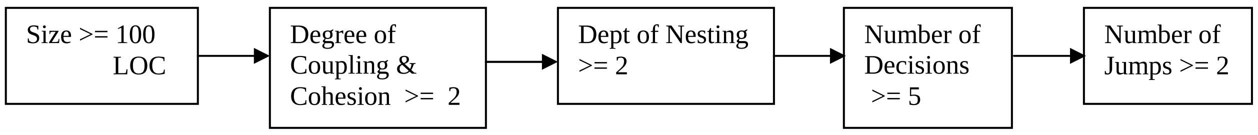
EEPorterS tree assumes that, there was enough time for the developers to code the software (i.e. the software project was not done in a haste) and that the coding was done by experience programmers/developers using familiar software development tools. In order to use the tree, for each software component or module, the following parameters have to be obtained: module size, degree of coupling, dept of nesting, number of decisions and number of jumps. Using these parameters to traverse the tree, a module is likely to meet one of the five conditions, which in turn determine its rank and test sequence.

7. Module Ranks and Components Attributes

The tree presents a software tester with five possibilities resulting from the attributes of software components. The possibilities represent the different ranks of program modules with respect to fault availability. Module ranks and components attributes are given below:

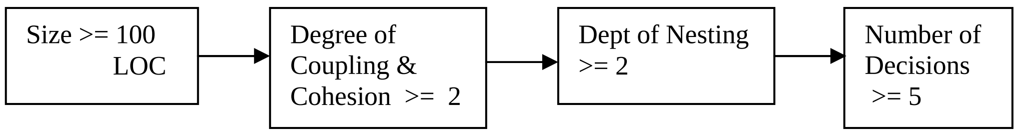
a) Rank 1 Module

Rank 1 module refers to a module that is likely to harbor the highest number of harmful errors than modules of other ranks. Such a module should be diagnosed and treated for residual errors before modules of other ranks. Any program module in software that satisfies the following condition is termed Rank 1 module.



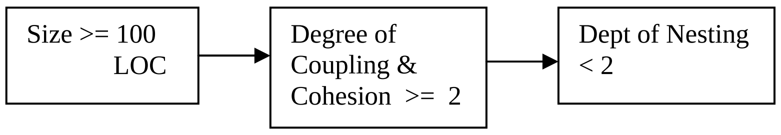
b) Rank 2 Module

Modules that satisfy the following conditions fall under Rank 2. They are expected to have fewer errors than Rank 1 module.

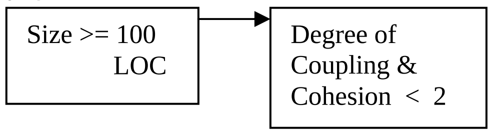


Conditions for Ranks 3, 4 and 5 are given below. The higher the module ranks the fewer the errors.

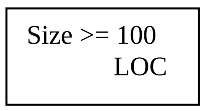
c) Rank 3 Module



d) Rank 4 Module



e) Rank 5 Module



8. Application of EEPorterS Tree

To illustrate the workability of EEPorterS Tree, 20 projects were tested using two methods, namely Random Testing and EEPorterS Testing. In the case of the later, EEPorterS Tree was used to rank the modules in each of the 20 projects. Module Ranks and test sequence are shown in table 1 while test records for each of the 20 projects comprising of test hours and errors found based on the two test methods are listed in table 2.

Table 1: Module ranks and Test Sequence

Project	Num. of Modules	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5
1	52	25, 26, 27, 35, 36, 37, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 51, 52	15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 28, 29, 30, 38, 39	11, 12, 13, 14, 31, 32, 33, 34, 47, 50	4, 5, 6, 7, 9, 10	1, 2, 3, 8
2	65	20, 21, 22, 29, 30, 31, 32, 41, 42, 43, 44, 45, 46, 47, 50, 51, 52, 53, 60, 61, 62, 63, 64	13, 14 15, 16, 17, 18, 19, 23, 24, 25, 26, 27, 28, 38, 39, 48, 49, 65	11, 12, 33, 34, 35, 36, 37, 54, 55, 56, 57, 58, 59	7, 8, 9, 10, 40, 46	1, 2, 3, 4, 5, 6
3	62	20, 21, 22, 32, 33, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 59, 60, 61, 62	17, 18, 19, 23, 24, 25, 26, 27, 28, 29, 30, 31, 38, 39, 54, 55, 56	5, 6, 12,13, 14, 15, 16, 34, 35, 36, 37, 57, 58	3, 4, 7, 8, 46	1, 2, 9, 10, 11
4	58	7, 8, 9, 10, 24, 25, 26, 27, 28, 29, 30, 36, 37, 40, 41, 42, 43, 52, 53, 54, 55, 56, 57, 58	11, 12, 13, 14, 15, 17, 18, 19, 21, 22, 23, 26, 28, 29, 38, 39	31, 34, 44, 45, 47, 48, 49, 50, 51	16, 32, 33, 35, 46	1, 2, 3, 4, 5, 6, 20
5	63	25, 26, 27, 28, 29, 30, 31, 32, 33, 41, 42, 43, 44, 45, 46, 47, 60, 61, 62, 63	15, 17, 18, 19, 20, 21, 22, 23, 24, 36, 37, 38, 39	34, 48, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59	3, 4, 8, 9, 10, 11, 12, 35, 40, 49	1, 2, 5, 6, 7, 13, 14, 16
6	52	15, 16, 17, 18, 19, 20, 27, 28, 29, 30, 36, 37, 40, 41, 50, 51, 52	23, 24, 25, 26, 38, 39, 42, 43, 44, 45, 46, 47	11, 12, 13, 14, 31, 34, 48, 49	7, 8, 9, 10, 32, 33, 35	1, 2, 3, 4, 5, 6, 21, 22
7	62	20, 21, 22, 27, 28, 29, 30, 40, 41, 48, 49, 50, 51, 52, 55, 56, 57, 58	12, 13, 14, 15, 26, 38, 39, 42, 43, 44, 45, 53, 54, 61, 62	16, 17, 18, 19, 31, 34, 35, 36, 37	9, 10, 11, 23, 24, 25, 32, 33, 35	1, 2, 3, 4, 5, 6, 7, 8, 46, 47, 59, 60
8	54	25, 26, 27, 28, 29, 30, 36, 37, 44, 45, 46, 47, 48, 49, 50, 54	6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17	31, 32, 33, 34, 35, 38, 39, 40, 41	20, 21, 22, 23, 24, 51, 52, 53	1, 2, 3, 4, 5, 18, 19, 42, 43
9	42	9, 10, 11, 12, 13, 14, 15, 18, 25, 26, 27, 30, 36, 37, 40, 41, 42	17, 19, 20, 21, 22, 23, 24, 28, 29, 38, 39	31, 33, 34, 35	3, 4, 16, 32,	1, 2, 5, 6, 7, 8,
10	52	15, 16, 17, 18, 19, 20, 26, 27, 28, 29, 30, 40, 41, 48, 49, 50, 51, 52,	12, 13, 14, 21, 22, 38, 39, 42, 43, 44, 45, 46, 47	31, 32, 33, 34, 35, 36, 37	7, 8, 9, 10, 11, 23, 24, 25	1, 2, 3, 4, 5, 6
11	61	20, 21, 22, 23, 24, 25, 32, 33, 34, 35, 39, 49, 50, 51, 52, 53, 57, 58, 59, 60, 61	16, 17, 18, 19, 41, 46, 42, 43, 44, 45, 46, 47, 48, 54, 55, 56	10, 11, 12, 13, 14, 15, 26, 27, 28, 29, 30, 31, 40	3, 4, 5, 6, 8, 9	1, 2, 7, 36, 37, 38
12	47	18, 19, 20, 25, 26, 27, 28, 29, 30, 36, 37, 45, 46, 47,	8, 9, 10, 11, 12, 13, 14, 15, 16, 17	31, 32, 33, 34, 35, 38, 39, 40, 41	21, 22, 23, 24, 51, 52, 53	1, 2, 3, 4, 5, 6, 7, 42, 43, 44
13	62	15, 16, 17, 18, 19, 20, 23, 24, 25, 26, 27, 28,	42, 43, 44, 45, 46, 47, 48, 49,	7, 9, 10, 11, 12, 13, 14, 21,	32, 33, 35, 38, 39,	1, 2, 3, 4, 5, 6,

		29, 30, 36, 37, 59, 60, 61, 62	50, 51, 56, 57, 58	22, 31, 34, 52, 53, 54, 55	40, 41	8
14	51	26, 27, 28, 29, 30, 31, 32, 33, 39, 40, 41, 47, 48, 49, 50, 51	13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 42, 43	23, 24, 25, 34, 35, 36, 37, 38, 44, 45, 46	5, 6, 7, 8, 9, 10, 11, 12	1, 2, 3, 4
15	36	15, 16, 20, 21, 22, 23, 24, 25, 26, 27, 28, 36,	7, 8, 9, 10, 11, 12, 13, 14,	17, 18, 19, 31, 32, 33, 34, 35	29, 30	1, 2, 3, 4, 5, 6
16	67	20, 21, 22, 24, 32, 33, 34, 35, 39, 49, 50, 51, 52, 53, 55, 61, 64, 65, 66, 67	16, 17, 18, 36, 37, 38, 46, 42, 43, 44, 45, 46, 47, 48, 56	8, 9, 10, 12, 13, 14, 15, 25, 26, 27, 28, 29, 31, 54, 57, 58	3, 4, 5, 6, 11, 19, 23, 30	1, 2, 7, 40, 41, 59, 60, 62, 63
17	48	10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 28, 29, 30, 36	21, 22, 37, 38, 39, 42, 43, 44, 45, 46, 47, 48	23, 25, 26, 27, 31, 34, 40, 41	5, 6, 7, 8, 9, 32, 33, 35	1, 2, 3, 4
18	56	30, 31, 32, 33, 34, 35, 39, 40, 41, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56	16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 42, 43, 44	10, 11, 12, 26, 27, 28, 29, 45, 46	5, 6, 7, 8, 9, 13, 14, 15	1, 2, 3, 4, 36, 37, 38
19	62	10, 11, 12, 13, 14, 15, 16, 17, 18, 25, 26, 27, 34, 35, 36, 37, 50, 51, 52, 53, 54	19, 20, 21, 22, 23, 24, 28, 29, 30, 31, 38, 39, 40, 41, 42	16, 17, 46, 47, 48, 49, 55, 56, 57, 58, 59, 60	3, 4, 5, 7, 8, 9, 32, 33, 43, 44, 45	1, 2, 5, 6, 61, 62
20	65	25, 26, 27, 28, 36, 37, 43, 44, 45, 46, 47, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62	10, 11, 12, 13, 14, 15, 16, 20, 21, 22, 23, 24, 41, 42, 48, 49, 50	17, 18, 19, 31, 33, 34, 35, 63, 64, 65	16, 29, 30, 32, 38, 39, 40	1, 2, 3, 4, 5, 6, 7, 8,

Table 2: Test Records from the two methods

Project	LOC	Random Testing		Testing with EEPorterS Tree		Difference in Test Hours (A-B)
		Test Hours (A)	Errors Found	Test Hours (B)	Errors Found	
1	12,700	316	72	89	115	227
2	10,150	295	64	101	87	194
3	8,720	354	23	87	71	267
4	12,220	276	95	78	107	198
5	10,220	325	54	85	91	240
6	12,120	368	67	89	103	279
7	10,840	352	43	103	101	249
8	10,620	293	52	76	95	217
9	12,200	311	89	111	105	200
10	12,500	382	84	88	115	294
11	11,020	374	91	91	97	283
12	8,200	268	47	105	79	163
13	12,785	271	83	74	118	197
14	12,130	282	115	68	118	214
15	11,620	321	72	73	107	248
16	10,120	335	28	102	89	233
17	11,630	284	18	78	87	206
18	10,710	361	53	65	95	296
19	12,120	288	112	83	112	205
20	10,810	325	93	69	95	256
Total					**1275**	**4666**
Average					**64**	**233**

9. Data Analysis and Results

Test records from the two methods were analyzed using LOC models to obtain total residual errors in each project, error difference and percentage error difference. The results of the analysis are presented in table 3.

Table 3: Random Testing Vs. EEPorterS Tree Testing

Project	LOC	Total Resid-ual Errors	Random Testing			Testing with EEPorterS Tree			Error Differ-ence	% Differ-ence
			Test Hours	Errors Found	Errors Rema-ning	Test Hours	Errors Found	Errors Rema-ning		
1	12,700	127	316	72	55	89	115	12	43	78
2	10,150	101	295	64	37	101	87	14	23	62
3	8,720	87	354	23	64	87	71	16	48	75
4	12,220	122	276	95	27	78	107	15	12	44
5	10,220	102	325	54	48	85	91	11	37	77
6	12,120	121	368	67	54	89	103	18	36	67
7	10,840	108	352	43	65	103	101	7	58	89
8	10,620	106	293	52	54	76	95	11	43	80
9	12,200	120	311	89	31	111	105	15	16	52
10	12,500	125	382	84	41	88	115	10	31	76
11	11,020	110	374	91	19	91	97	13	6	32
12	8,200	82	268	47	35	105	79	3	32	91
13	12,785	128	271	83	45	74	118	10	35	78
14	12,130	121	282	115	6	68	118	3	3	50
15	11,620	116	321	72	44	73	107	9	35	80
16	10,120	101	335	28	73	102	89	12	61	84
17	11,630	116	284	18	98	78	87	29	69	70
18	10,710	107	361	53	54	65	95	12	42	78
19	12,120	121	288	112	9	83	112	9	0	0
20	10,810	108	325	93	15	69	95	13	2	13

10. Results Interpretation

a) Test Hours

The table clearly shows that with EEPorterS, fewer test hours were used to detect more errors compared to random testing approach. From table 2, on the average for the 20 projects, EEPorterS test hours were reduced by 233 hours (approximately 10 days), which is great improvement. This implies that, with EEPorterS software test hours can be reduced even as more errors are detected and corrected. The 233 hours could be directed to other projects rather than being wasted in redundant testing of ongoing projects.

b) Found Error Difference

Table 3 shows that, in terms of the difference in errors found by the two methods with the 20 projects, testing with EEPorterS Tree records 64% improvement on the average over the Random test method. For project 19, found error difference of 0 was recorded which implies that equal number of errors were detected by the two methods. For the 20 projects, there was no case of more errors detected by the random testing, hence EEPorterS is better.

11. Recommendations

Based on the outcome of this research, we wish to make the following recommendations:

1. In order to minimize redundant testing of a software, EEPorterS tree should be used to quickly identify fault-prone modules/components for diagnosis and treatment.
2. Further research in this area is recommended for interested researchers with greater emphasis on how to automate EEPorterS.
3. Software Testing and debugging should be given serious attention by upcoming software organizations and programmers to produce quality software products.
4. Governments should encourage the development of quality and reliable software that will meet international standards through proper funding of research works in software error detection and diagnosis and favorable policies/incentives to promising software organizations.

12.Conclusion

Testing and debugging are very important exercises during integration process and require serious attention to ensure that the software meets the acceptance criteria specified in the software contract even as the software is released at the optimum release time. Because, software reliability, availability and even maintenance costs depend on the quality of testing and debugging performed, specialized tools are required to identify modules that are likely to harbor much harmful errors. However, since software errors are no physical objects that can be easily identified in programs, programmers most times spend much time finding bugs in modules where they do not exist. Having a tool like EEPorterS tree will enable testers to quickly identify modules likely to harbor much harmful errors for diagnosis and treatment rather than searching for errors aimlessly.

Once the tester is able to rank (i.e. Rank 1 to Rank 5) the modules in the software using EEPorterS tree, testing can begin with Rank 1 modules followed by Rank 2 in that order. The number of found errors and test hours will reduce considerable as the test progresses from Rank 1 modules to Rank 2 modules and so forth. Therefore, the research indicates that the higher the rank the lower the errors likely to reside in the module. With EEPorterS, since testing and debugging are likely to start with error-prone modules, errors are likely to be detected and corrected at a faster rate thereby making it possible for all modules to be properly tested and the software released within the stipulated time.

ISSN 1512-1232

13. References

[1] Avison, D. E. and Firtzland, G. Information System Development, Methodologies, Techniques, and Tools. Blackwell Scientific Publications, London, 1998, pp. 237-315.
[2] Fairley, E. Software Engineering Concept. McGraw-Hill Book Company, Singapore, 1985, pp. 310-339.
[3] Shooman, M. L. Software Engineering, McGraw-Hill Book Co,Singapore, 1983, pp. 469_490.
[4] Graham, D., Veenedaal, E. V., Evans, I. And Black R. Foundations of Software Testing, ISTQB Certification, Thomas Learning, High Holborn House, 50-51 Bedford Row, London, 2007, pp. 3 and 156
[5] Flanagan, C. and Qadeer, S. *Predicate Abstraction for Software Verification*, Proceedings of the 29th ACM SIGPLA-SIGACT Symposium on Principles of Programming Languages, Vol. 37, 2002.
[6] Brand, D. *A Software Falsifier*, International Symposium on Software Reliability Engineering, 2002, pp. 174-185.
[7] Pfleeger, S. L. Software Engineering, Theory and Practice, Second Edition, -Hill Companies, New York, 1997, pp. 372-378.
[8] Pressman, R. S. Software Engineering, A Practitioner’s Approach, Sixth Edition, McGraw-Hill Companies, New York, 2005, pp. 442-443, 447-453
[9] Wattam, S.I. Software Engineering, a Dynamic Approach, Sigma Press, Wilmslow, England, 1991, pp. 52,135-138.
[10] Alain, V. Reliability, Availability, Maintainability and Safety Assessment. John Wiley and Sons, New York, 1992, pp. 461-497.
[11] Buchner, F. The Classification Tree Method. Hitex development Tools, Karlsruhe Companies, Germany, 2002.

Article received: 2010-01-27