Send Orders of Reprints at reprints@benthamscience.net

An Empirical Study of Different Types of Changes in the Eclipse Project

Pitamber Tiwari, Wei Li, Raouf Alomainy* and Bingyang Wei

Computer Science Department, University of Alabama in Huntsville, Huntsville, Alabama 35899, USA

Abstract: This paper studied the distribution of different types of changes in the various contexts of the system and the relationship between artifact (file and module) size and different changes. We used the change data in the open source Eclipse Project through its decade-long evolution history. The latest release has 220 modules, 33904 files, 3780201 lines of code, and 49853 changes (accumulatively). This study focused on two levels of software artifacts: module and file; and four contexts of changes: all changes, error changes, non-error changes, and 19 change categories.

At the module level, we found that the power-law distribution was a common phenomenon for three contexts of changes at both the module and file levels: it existed in all changes, in error changes, and in non-error changes. When we analyzed the 19 change categories, the files and modules exhibited different behavior: the power-law distribution existed in all but one category at the module level, but, about two third of the change categories did not show the power-law distribution at the file level.

On the relationship between artifact size and changes, we found, at the module level, a few modules that had the majority of changes accounted for the majority of the code size; however, this phenomenon disappeared when we separated the error from non-error changes. At the file level, this phenomenon did not exist at all. We did not find any correlation between artifact size and changes at either the module or file level.

Keywords: Software change distribution, large object-oriented system, Post-release evolution.

1. INTRODUCTION

Making changes to a large object-oriented software system is a common, difficult, and error-prone process that many software engineers face daily. Software changes are triggered by two types of events: to fix errors (bugs) and to make changes for non-error reasons, such as refactoring [1]. Understanding change characteristics in software systems, especially where they concentrate at various levels of artifacts, can help us find effective methods to handle changes.

1.1. Software Errors

Software error (bug) distribution has been investigated by many studies. Basili and Perricone found that the majority of errors were due to incorrect functionality specifications or requirements. They also found that 62% of the changes were due to error correction and 38% were due to modifications [2]. Shen and colleagues discovered the inverse relationship between size and number of errors up to the size of 500 lines [3]. Withrow found that the error density rose with respect to size beyond 255 lines and suggested that the testing should focus on modules of size either less than 200 lines or greater than 500 lines [4]. Moller and Paulish discovered that the changes in the source code increased error rate and the percentage of changed code and number of errors had inverse relationship [5]. Ohlsson and Alberg showed that 60% of the errors were located in 20% of the modules [6]. A study conducted by Hatton showed that the errors were concentrated in modules with 200 to 400 lines of code (LOC) [7]. Fenton and Ohlsson showed that errors followed "Pareto principle of distribution" but did not find LOC as a good predictor [8]. The study conducted by Ostrand and colleagues showed support for Pareto principle in errors and discovered LOC as the strongest predictor [9]. Andersson and Runeson showed that errors followed "Pareto principle of fault distribution" and size metrics were good predictor of pre-release errors in modules [10]. Zhang discovered that Weibull probability distribution function described the error distribution more precisely [11]. El Emam and colleagues studied a large C++ system and found the confounding effect of class size on the prediction of software faults, and suggested that class size was a major predictor of class faults [12].

These studies suggested that Pareto principle was a common phenomenon for software errors, whereas the relationship of artifact (class, file, or module) size and errors was less clear. Our research objective is to verify the results from the previous error distribution studies by using the data from a large open source system: Eclipse, in its decade long evolution history.

1.2. Error Categories

Although there have been many studies on software errors, few have analyzed software errors in detailed categories. The Institute of Electrical and Electronics Engineers (IEEE) published the Standard Classification for Software Anomalies [13] that introduced error categories. Other error

*Address correspondence to these authors at the Computer Science Department, University of Alabama in Huntsville, Huntsville, Alabama 35899, USA; Tel: +1 (256) 509-1468; Fax: +1 (256) 8240-6239;
E-mail: ralomain@cs.uah.edu

1874-107X/13

classifications were introduced by Li and colleague [14], Lo and colleagues [15], and Pan and colleagues [16] .

Previous work has focused on error vs. no error. Our research objective is to classify errors into more detailed categories and verify the research results from previous studies. Knowing where each type of error concentrates will help us develop more focused methods that avoid introducing them or find a certain type of errors in software. For example, if we know that the memory-related errors: memory leaks, dangling pointers, and array-out-of-bounds, are concentrated in a small set of modules that share certain characteristics, we can develop tools that monitor the module characteristics and alert programmers to watch out for memory-related errors when these modules are being developed or maintained, thus reducing memory errors in code. This knowledge also allows us to be more precise about where and how to find certain types of errors. For example, we can increase the testing effort to hunt for memory errors when those modules are identified and tested.

1.3. Non-Error Changes

Often, software engineers make changes to software systems for non-error reasons (such as refactoring). The studies on changes that were not caused by fixing errors were scarce. Understanding where non-error changes concentrate is not less important to that of errors, because changes, regardless of the triggers, are prone to errors and affect software quality. Soares and colleagues in their study [17] presented a technique to identify issues, mainly on semantics, which resulted from non-error refactoring effort. Görg and Weißgerber showed how incomplete refactoring can cause long standing bugs [18].

1.4. Is Software Size an Effective Predictor of Error Rate

Many previous studies investigated whether the size of a module is an effective predictor of the module's error rate [5, 9, 10, 12, 19, 20]. The majority of these studies found the two correlated. Our research goal in this aspect is to investigate if a correlation between the two exists in the large open source system's evolution history.

1.5. The Scope of this Research Study

In this study, we have investigated the power-law distribution (Pareto principle) in many contexts: at module level, a file level, for all changes, for error changes, for non-error changes, and for each of the 19 change categories. We also examined the relationship between changes and artifact size in the same contexts. We studied 220 modules, 33,904 files, and a total of 3,780,201 lines of code (LOC) - the total number of lines in the source code excluding the comments and blank spaces - and 5,908,044 physical lines - the total lines in the source code including the blank lines, single braces or parentheses, - and a total of 49,853 unique change identifiers in the Eclipse Project's evolution history that spanned more than a decade.

1.6. Research Objectives

The objective of this research is twofold: to study the change distribution and to investigate the relationship between changes and artifact size. We are interested in verifying the Pareto principle in change distribution, that is, a small percentage of artifacts (modules and files) contain the majority of changes (all changes, error changes, non-error changes, and each category of changes). The following are the research hypotheses:

Hypothesis #1: Changes have the power-law distribution at the module and file level.

Hypothesis #2: Both error changes and non-error changes follow the power-law distribution.

Hypothesis #3: Changes in each classification category follow the power-law distribution.

Hypothesis #4: The artifact size is correlated with changes in the artifact.

Hypothesis #5: Modules and files with majority of changes contain most of the code size.

2. MATERIALS AND METHODOLOGY

Eclipse is a large composite open source software system that is made of many subsystems, primarily written in the Java programming language. Each subsystem is made of several projects. Out of several Eclipse subsystems, we focused on the Eclipse Project subsystem. Eclipse Project consists of five projects: e4, Eclipse Project Incubator (EPI), Java Development Tool (JDT), Plugin Development Environment (PDE), and Eclipse Platform (EP). JDT, PDE, and EP are further divided into subprojects. Each subproject comprises of *modules* and *module alias*. A module is made of source files. A module alias is a virtual module that is made of source files that come from other modules. For example, Module A has two source files: A1.java and A2.java, and Module B has two source files: B1.java and B2.java. A module alias MA may be made of A1.java and B2.java. Every project or sub-project under Eclipse Project is a combination of several modules and/or module aliases. We only analyzed modules in this study because module alias did not contribute any new files to the system.

2.1. The CVS and Bugzilla Repositories

Eclipse uses Concurrent Versioning System (CVS) repository to keep track of changes in project files [21]. CVS is a version control system that maintains history of the files throughout their development and evolution.

We created a crawler tool to extract information from the Java files located in the Eclipse CVS repository. To classify each error change to a specific error category, we tried the *change description* field from the CVS repository first. The change descriptions (known as *bug descriptions* in CVS and Bugzilla) extracted from the CVS repository were the titles of the changes. They did not have enough information to describe a change completely for classification purpose. On several runs, we found the change descriptions to be misleading and inappropriate, sometimes even empty. To mitigate the problem of insufficient information in the CVS repository, we went to Bugzilla, another change repository for the Eclipse system, which had more detailed change description for each change identifier (bugID) that we found in the CVS repository. The crawler tool extracted the module name, file name, revision, bug number (bugID), and bug description from the CVS repository and then used the bugID to extract the detailed description of the changes from the Bugzilla.

2.2. Change Categories

We classified the changes into 19 specific categories and studied the change distribution in each category. The change categories that we adopted came from the combination and customization of the categories suggested and used by Li and colleagues [14], Pan and colleagues [17], Lo and colleagues [15], and IEEE Standard Classification for Software Anomalies [13]. Table **1** summarizes the change categories that we used in our study.

Table 1. List of Change Categories

Bug Category	Description
Adaptive Maintenance	A type of maintenance performed to change software so that it will work in an altered environment, such as when an operating system, hardware platform, compiler, software library or database structure changes, compatibility changes, etc.
Enhancement	Actions performed to add a new capability or improve the existing capability to software; Bugs that cause failure to meet the performance requirements of the product. Examples: Functions correctly but runs/respond slowly. Takes longer time to perform a task. Addition of a new feature.
Not Bugs	Test cases, simple updates. Example: File name update, copy right update, and test case addition.
Refactoring	Changes of source code without modifying the functional behavior to improve some nonfunctional attributes, such as readability, reduced complexity, or maintainability, of the software.
Coding	Bugs in the decision logic, branching, sequencing, computational logic, and typos in programming. Examples: Wrong concept employed in coding. Navigation not coded correctly in source code. The control flow is incorrectly implemented. Incorrect processing and evaluation of expression and equations.
Concurrency	Bugs that happen in multithreading or multiprocess environment, including data race, deadlock and synchronization.
Data	Bugs in definition, structure, mapping, access, scope, use, or initialization. Examples: Incorrect object type or dimension. Incorrect initial default values. Incorrect duplication or failure to create a duplicate object. Incorrect access to object. Use of incorrect variable names.
Documentation	Bugs in the specifications and Java documents. Bugs in program help.
Functionality	Function is incorrect, ambiguous, inconsistent, missing, incomplete, that do not need to be present, or do not behave as expected. Bugs related to core functionality implementation. Functions that do not meet the specification requirement or the standards with the software version, or coding conventions, representation. Any other semantic bug that does not meet the design requirements. Examples: Unnecessary features. Duplicate features. Misplaced features. Missing Validation.
Generic	Unknown bugs. Bugs related to security. Bugs related to scalability. Bugs with missing bug description.
GUI	Bugs related to graphical user interface and CSS bugs. Examples: Incorrect alignment of components. Incorrect size and shape of interface. Incorrect layout of reports. Incorrect coloring.
Handling Event/Exception	Missing event handling or improper event handling. Do not have proper exception handling. Anomalies caused by exception. Examples: Null pointer exception. Class cast exception. Missing key press or mouse movement handling. Incorrect mouse click, hover, and double click handling. Incorrect action selection. Timer expiration.
I/O, Serialization	Bugs related to I/O handling, import and export of files, serialization of objects. I/O–related to read from and write to files, memories, sockets.
Internationalization: Localization with Resource Bundles	Bugs due to adaption to various languages and regions without engineering changes. Bugs due to adaption of internationalized software for a specific region or language.
Memory	Bugs related to memory. Examples: Buffer overflow. Memory access violations. Memory leak. Dangling pointer. Null pointer dereference Double free.
Message	Defects in message and logs, inadequate, incorrect, misleading, or missing bug messages in source code or bug logs.
Regression/rollback	Bugs due to changes in some function or code, change rollback, and changes not made

	properly. Examples: Incorrect merging of codes. You commit a fix, but it did not work so the changes have to be rolled back.
Third Party	Bugs due to third party software.

The top four categories are non-error changes, whereas the remaining categories are all error categories.

2.3. Classification and Conflict Resolution

Change categorization or classification is a difficult task. Sometimes, a change could be classified into several different categories. There seems to be no universally correct way to categorize changes to a particular change category. Change categories, literally, can be infinite [22].

We tried the automated text (change) classification techniques of the Support Vector Machine [20] and the Naïve Bayes [19] and the tool provided by Dr. Paul Wofgang [23], and found that the classification accuracy was not acceptable. Therefore, we performed the change classification manually. Based on the history information, change description, comments, reviews from developers, each change (bugID) was placed in only one category. We tried to resolve the conflicts based on the conscience of the comments in Bugzilla. When a bug could not be placed in a known category, it was placed in the *Generic* category. When a non-error change could not be placed in any of the three clearly defined non-error categories, it was placed in the *Not Bugs* category.

Out of the 220 modules that we examined, 52 modules were change free. We classified the changes in 140 modules, which accounted for 83.3% of the modules that had changes.

3. ARTIFACT SIZE AND CHANGES

To analyze the relationship between artifact size and changes, we used *Resource Standard Metrics (RSM)*, a source code metrics and quality analysis tool, to get the size metrics. For our study, we used the tool's default source code size metrics for modules and files. The source code size metrics available from the tool are the *Lines of Code (LOC), Effective Lines of Code (eLOC), Logical Lines of Code (lLOC), Comment,* and *Lines.*

LOC is the total number of lines in the source code excluding the comments and blank lines.

eLOC gives the count of the total number of source code lines that are not comments, blank lines, standalone parenthesis or braces. The eLOC metric is a result of subtracting total number of single braces or parenthesis from the LOC metric. Many programmers put single brace or parenthesis on a new line to make source code clean and readable. This programming style leads to the increase of the LOC metric, but not eLOC.

lLOC represents the total number of lines in the source code that are terminated by a semicolon. The 'for' loop control statement contains two semicolon, however it accounts for only one in the calculation of the lLOC metric.

Comment represents the total lines of comments in the source code. This metric is a general measure of the effort by the developer to make the program more understandable.

Lines is the total lines in the source code including the blank lines, single braces, or parenthesis. It is sometimes called *Physical Lines of Code*.

3.1. Metric Calculation

We use an example, adapted from the documentation of RSM, to show how the RSM tool calculated the metrics. In Table **2**, the presence of check mark (✓) indicates the particular line in the sample source code is included in the count of the respective metrics. The table indicates that the tool considered four lines of source code for the calculation of *LOC*, two lines of source code for the calculation of *eLOC*, one line of source code for the calculation of lLOC, two lines of source code for the calculation of Comment, and six lines of source code for the calculation of Lines.

4. RESEARCH DESIGN

To test the hypotheses, we collected the change data from 220 modules that contained 33,904 files from the Eclipse Project. The number of changes ranged from 0 to 6365 per module and from 0 to 202 per file. The total number of changes in modules and files was 49,853. Modules and files had sizes ranging from 20 LOC to 405,951 LOC per module and 1 to 48,596 LOC per file.

The power-law distribution [24] occurs in many situations of software development and evolution. The study conducted by Potanin and colleagues [25] illustrated the power-law distribution in the incoming and outgoing references in object graphs. Wheeldon and Counsell [26] confirmed the power-law distribution in object-oriented class relationships. Hatton [27] illustrated the power-law distribution in the equi-

Table 2. An Example Showing the Process Used by RSM to Determine the Size Metrics

Source Code	LOC	eLOC	lLOC	Comments	Lines
if(score>90) //if statement test condition	✓	✓		✓	✓
{	✓				✓
//assign the grade				✓	✓
					✓
grade = 'A';	✓	✓	✓		✓
}	✓				✓

Table 3. Distribution of Change Percentage Over Module Percentage

# of Modules	% of Modules	# of Changes	% of Changes
11	5	30505	61.19
22	10	40275	80.79
33	15	43997	88.25
44	20	45838	91.95
55	25	47138	94.55
66	30	47949	96.18
77	35	48572	97.43
88	40	48975	98.24
99	45	49268	98.83
110	50	49475	99.24
121	55	49626	99.54
132	60	49746	99.79
143	65	49806	99.91
154	70	49835	99.96
165	75	49850	99.99
168	76	49853	100.00

librium component size of a system. These studies suggested that the power-law distribution was a common phenomenon in software systems.

Relatively small numbers of modules or files are believed to have most of the bugs in software systems [5, 7, 8, 10]. This idea can be summed up by the Pareto Principle [28], also known as the 20-80 rule. The power-law and Weibull distribution are commonly used distributions to confirm the Pareto Principle.

We used the Alberg Diagram, as suggested by Fenton and Ohlsson [8], and used by Timea and Barbara [29] and Anderson and Runeson [10], for testing Hypothesis #1. We sorted the modules and files in decreasing order with respect to the number of changes. We plotted the accumulative percentage of changes on the y-axis of the Alberg Diagram and the accumulative percentage of the modules or files on the x-axis. The Alberg Diagram provides the visual identification of a possible power-law distribution. To be certain, we used the function built in Matlab [30] as described by Aaron and colleagues [31], to test the power-law distribution at the module and file level.

To test Hypothesis #4, we calculated the Pearson correlation coefficient [24] between the size metrics and the changes at the module and file level.

To test Hypothesis #5, we computed and analyzed the sizes of modules and files that were responsible for a large percentage (80% for example) of the changes. We sorted the modules and files in decreasing order relative to the number of changes, and then plotted the accumulative percentage of changes and the LOC metric on the y-axis with respect to the accumulative percentage of modules or files on the x-axis to see if the two curves were close.

5. DATA ANALYSIS RESULTS

In this section, we present the results of the hypothesis testing. In testing Hypothesis #1, we show both the visual plots and the statistical tests to illustrate how we analyzed the data. In subsequent hypothesis tests, we only show the summary of the statistical tests to save the space.

Hypothesis #1: Changes have the power-LAW distribution at the module and file level.

We collected 49,853 bugIDs (the unique identifiers for the changes) that affected 220 modules in the Eclipse Project. The modules were sorted in descending order with respect to the number of changes. Table **3** summarizes the distribution data for the modules.

5.1. The Alberg Diagram for Modules

Fig. (**1**) shows the Alberg Diagram for the distribution data in Table **3**. The accumulative percentage of changes is on the y-axis and the accumulative percentage of modules is on the x-axis.

At the module level, 20% of the modules were responsible for about 92% of the changes, and 76.36% of the modules were responsible for almost100% of the changes. Roughly a quarter of the modules did not report any changes. Fig. (**1**) shows the existence of the Pareto principle and the possibility of the power-law distribution.

5.2. Power-law Distribution for Modules Using the Statistical Tool

Aaron and colleagues provided several Matlab functions to test the power-law distribution [30]. We used the *plfit(x)* function, which estimated the lower cutoff value *xmin* and α

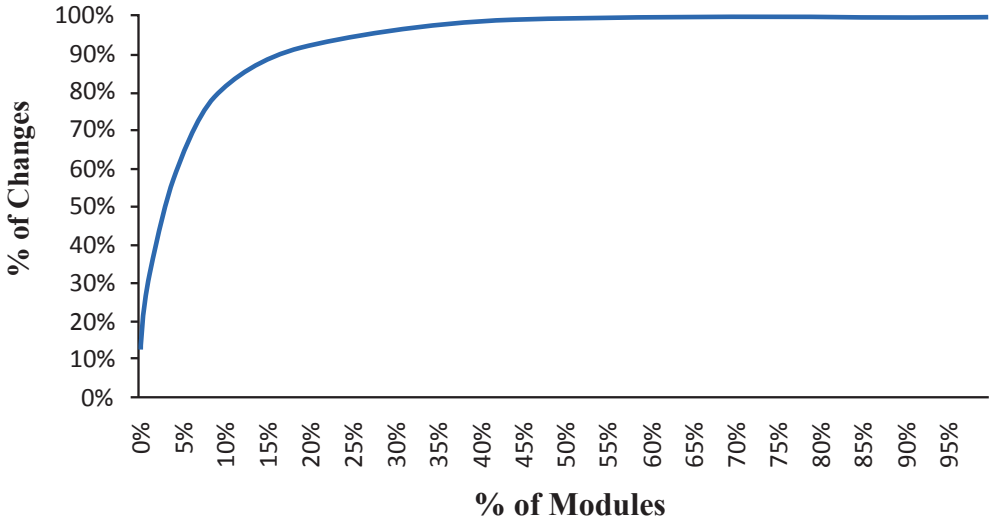

Fig. (1). Percentage of modules versus percentage of changes.

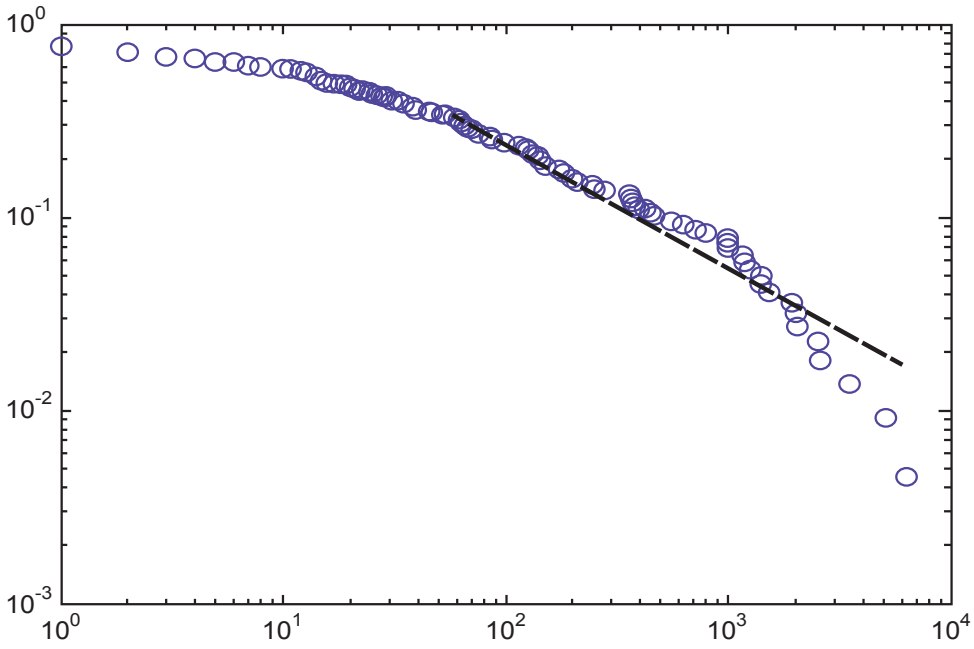

Fig. (2). Plot of the number of changes per module and power-law distribution for x greater than or equal to xmin.

(alpha) by implementing the maximum likelihood estimator and the goodness-of-fit. This function used the number of changes (x), given in Table **3**, as input. The calculated lower cutoff value (by the function) xmin was 53. To visualize the fitted distribution, we used the *plplot(x, xmin, alpha)* function, where x is the number of changes per module and xmin and α are the values computed by the *plfit* function. This function plotted the data and the fitted power-law distribution in log axes in Fig. (**2**), where the dashed line is the power-law distribution and the circled line is the module data. Fig. (**2**) shows that most of the module data fell on the line given by the power-law distribution.

We used the function *plpva(x, xmin)* that took the number of changes in modules (x) and lower cutoff value xmin to calculate the *p-value* for the Kolmogorov-Smirnov test [24]. This function estimated the p-value over many repetitions. The default number of repetitions was 1000, which was the number that we used. The p-value computation was a slow process. In our case, it took hours to compute one p-value. The time was dependent on the number of samples and the value of each sample. The p-value, for 1000 repetitions, computed by the function for the number of changes in modules was 0.049. We used the threshold value of 0.05 suggested by Aaron and colleagues [30]. If the computed p-value is greater or equal to 0.05, the distribution follows the power law; otherwise, it does not follow. The p-value 0.049 was close enough to 0.05, so we treated it as being equal to 0.05 and accepted the Hypothesis #1 for the modules and concluded that the total changes (error and non-error changes together) followed the power-law distribution at the module level.

The p-value that we used thereafter is the p-value suggested by Aaron and colleagues [30] and the tools implementing their work. This p-value is different from the p-value commonly used to reject null form of a hypothesis.

Table 4. Distribution of Changes in Files

# of Files	% of Files	# of Changes	% of Changes
1696	5.00	30149	60.48
3391	10.00	38443	77.11
3871	11.41	39883	80.00
5086	15.00	42911	86.08
6781	20.00	45820	91.91
8476	25.00	47515	95.31
10172	30.00	49211	98.71
10814	31.90	49853	100.00

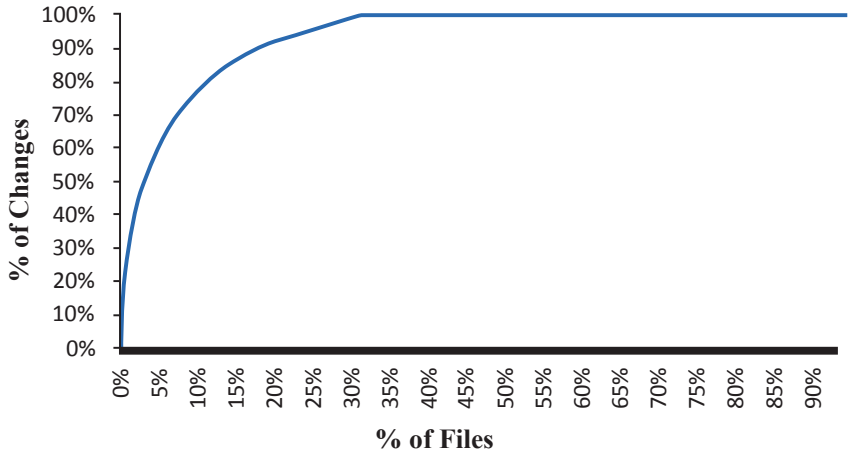

Fig. (3). Percentage of files versus percentage of changes.

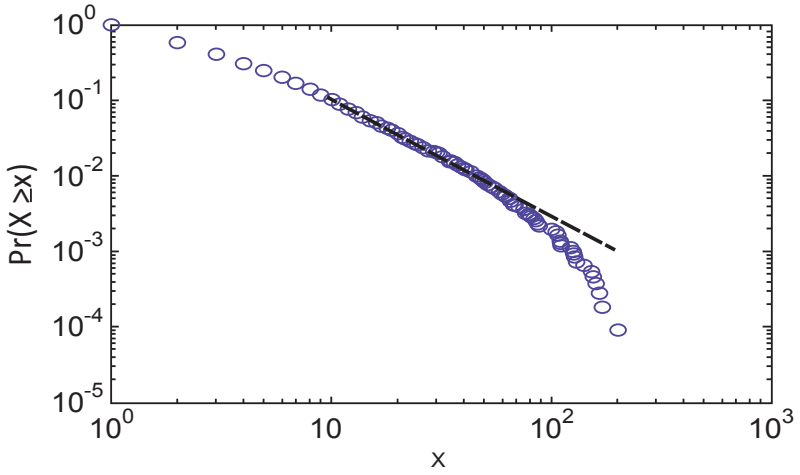

Fig. (4). Plot of number of changes per file and the power-law distribution for x greater than or equal to xmin.

5.3. The Alberg Diagram for Files

Table **4** summarizes the data for files. We did the same tests for files as we did for modules.

Fig. (**3**) shows the Alberg Diagram for the distribution of changes at the file level. At the file level, 20% of the files were responsible for more than 90% of the changes. Approximately 32% of the files had 100% of the changes; that is, about two-third of the files did not report any changes.

5.4. Power-Law Distribution for Files Using the Statistical Tool

Fig. (**4**) shows the power-distribution plot for the file data. Visually, we can see that the file data followed the power-law distribution line very closely. The calculated p-value was 0.106, which was greater than the threshold value of 0.05. Therefore, we accepted Hypothesis #1 for files and concluded that the total changes at the file level followed the power-law distribution.

Table 5. The p-Values for the Error and Non-Error Changes

p-value	Error Changes	Non-Error Changes
Modules	0.56	0.99
Files	0.11	0.45

Table 6. Distribution of Changes in Each Category at the Module Level

Category	P value	X min.	# of Modules Affected	Total Changes
Adaptive	0.032	5	50	474
Cleanup	0.738	2	43	553
Coding	0.312	13	59	667
Concurrency	0.925	5	32	542
Data	0.270	2	44	583
Doc	0.585	8	32	239
Enhancement	0.146	12	93	5459
Functionality	0.052	14	95	3236
Generic	0.176	5	86	1313
Gui	0.116	1	28	443
Handling	0.492	8	69	1285
I/O	0.167	8	3	255
Internationalization	n/a	n/a	31	10
Memory	0.254	6	28	254
Messaging	0.525	4	46	520
Not bug	0.788	20	81	1553
Refactoring	0.882	7	66	1751
Regression	0.409	2	40	272
Third Party	n/a	n/a	6	10

Since the changes at the module and file level followed the power-law distribution, we accepted the Hypothesis #1.

Hypothesis #2: Both error changes and non-error changes follow the power-law distribution.

To study the change distribution in each change category, we had to classify the changes into different categories. We tried an automated classification of the changes using data mining tools and the change descriptions that we collected from the CVS and Bugzilla repositories. Despite various attempts and different techniques, we could not get more than 65% accuracy of the automated classification. To ensure the quality of the classification and the confidence in the research results, we classified the changes manually. Out of the 49853 changes, we classified 19419 (about 40%) of them that affected 83.3% of the modules into 19 change categories. The manual classification was very time consuming, but yielded the most accurate classification possible. The changes that were classified were randomly chosen. Although we did not classify all the changes, we believe that enough changes were classified to give us an accurate account of the distribution. Of the 19 change categories, we merged all the non-error categories into one group and all the error categories into another. There were 9237 non-error changes and 10,182 error changes in modules, or 47.57% of the changes were non-error changes and 52.43% of the changes were error changes.

Table **5** summarizes the test results for the power-law distribution at the module and file level using the two groups. All p-values were greater than the threshold 0.05; therefore, we accepted Hypothesis #2 and concluded that error and non-error changes followed the power-law distribution at the module and file level.

Hypothesis #3: Changes follow the power-law distribution in each classification category.

Table **6** summarizes the p-values calculated for each change category at the module level; Table **7** is for files. The Matlab functions did not report the p-value for the Internationalization and Third Party categories, because there were less than 50 changes (50 was used as the cutoff value by the functions) in the two categories.

Table 7. Distribution of Changes in Each Category at the File Level

Category	P Value	X min.	# of Files Affected	Total Changes
Adaptive	0.019	1	343	474
Cleanup	0.044	1	404	553
Coding	0.000	1	431	667
Concurrency	0.028	1	283	542
Data	0.413	2	352	583
Doc	0.489	1	174	239
Enhancement	0.000	1	2481	5459
Functionality	0.608	4	1382	3236
Generic	0.634	3	815	1313
Gui	0.029	1	263	443
Handling	0.601	3	692	1285
I/O	0.000	1	164	255
Internationalization	n/a	n/a	10	10
Memory	0.023	1	173	254
Messaging	0.324	1	322	520
Not bug	0.000	1	1058	1553
Refactoring	0.326	3	952	1751
Regression	0.007	1	200	272
Third Party	n/a	n/a	9	10

Table 8. Spearman Correlation Coefficient Values for Modules and Files

Metrics	Modules	Files
LOC	0.63	0.36
eLOC	0.63	0.37
lLOC	0.63	0.38
Comment	0.65	0.42
Line	0.64	0.42

At the module level, all but one, Adaptive, categories had the p-value greater than 0.05. Therefore, we concluded that most but not all change categories followed the power-law distribution at the module level.

At the file level, close to two-third of the categories had the p-value less than 0.05; therefore, we concluded that most change categories did not follow the power-law distribution at the file level.

For Hypothesis #3, we obtained the mixed results: the modules and files exhibited different behaviors when changes were examined in each category. A detailed discussion on this difference is presented in the Discussion of the Findings section.

Hypothesis #4: The artifact size is correlated with the changes in the artifact.

Table **8** shows the Spearman correlation coefficient values between the LOC, eLOC, lLOC, Comment, and Lines metrics and the number of changes in the modules and files. These coefficients were very low and not statistically significant. Therefore, we rejected Hypothesis #4 and concluded that there was no correlation between artifact size and the changes.

Hypothesis #5: Modules and files containing majority of changes contain most of the code size (LOC).

5.5. LOC for Modules

In order to test this hypothesis, we calculated the LOC for each module using the RSM tool. There were 3,780,201 LOC across 220 modules. Table **9** summarizes the LOC data for the modules.

Table 9. Distribution of Accumulative Percentage of Changes and LOC for the Modules

# of Modules	% of Modules	# of Changes	% of Changes	LOC	% of LOC
11	5	30505	61.19	955419	25.27
22	10	40275	80.79	1970302	52.12
33	15	43997	88.25	2385225	63.10
44	20	45838	91.95	2569486	67.97
55	25	47138	94.55	3040524	80.43
66	30	47949	96.18	3135854	82.95
77	35	48572	97.43	3135931	85.00
88	40	48975	98.24	3136019	86.77
99	45	49268	98.83	3136118	88.14
110	50	49475	99.24	3136228	89.56
121	55	49626	99.54	3136349	90.50
132	60	49746	99.79	3136481	91.48
143	65	49806	99.91	3136624	93.44
154	70	49835	99.96	3136778	94.58
165	75	49850	99.99	3136943	95.32
168	76.36	49853	100.00	3137111	95.33

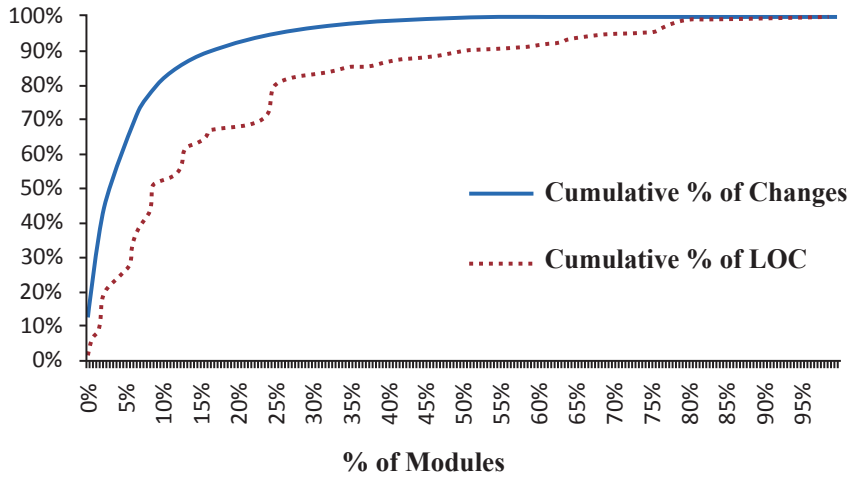

Fig. (5). Accumulative percentage of changes and LOC versus the accumulative percentage of modules.

Table 10. Distribution of Accumulative Percentage of Error Changes and LOC for the Modules

# of Modules	% of Modules	# of Error Changes	% of Error Changes	LOC	% of LOC
10	5.20	6677	65.58	879895	38.51
20	10.40	8073	79.29	1067303	46.72
30	15.60	8913	87.54	1465085	64.13
40	20.80	9417	92.49	1633979	71.52
50	26.00	9649	94.77	1701178	74.46
60	31.20	9818	96.43	1740594	76.19
70	35.88	9928	97.51	1788810	78.30
80	41.60	10036	98.57	1885703	82.54
90	46.80	10098	99.18	1914601	83.80

Table 10. contd…

# of Modules	% of Modules	# of Error Changes	% of Error Changes	LOC	% of LOC
100	52.00	10136	99.55	1996082	87.37
110	57.20	10162	99.80	2029362	88.82
120	62.40	10173	99.91	2085500	91.28
129	67.08	10182	100.00	2106066	92.18

Table 11. Distribution of Accumulative Percentage of Non-Error Changes and LOC for the Modules

# of Modules	% of Modules	#of Non-Error Changes	% of Non-Error Changes	LOC	% of LOC
10	5.20	6145	66.53	597260	26.14
20	10.40	7174	77.67	688207	30.12
30	15.60	7820	84.66	1020145	44.65
40	20.80	8286	89.70	1113032	48.72
50	26.00	8605	93.16	1352786	59.21
60	31.20	8811	95.39	1434835	62.80
70	36.40	8963	97.03	1460835	63.94
80	41.60	9068	98.17	1520795	66.56
90	46.80	9144	98.99	1895540	82.97
100	52.00	9198	99.58	1942151	85.01
110	57.20	9225	99.87	1968230	86.15
120	62.40	9235	99.98	2025147	88.64
122	63.44	9237	100.00	2026774	88.71

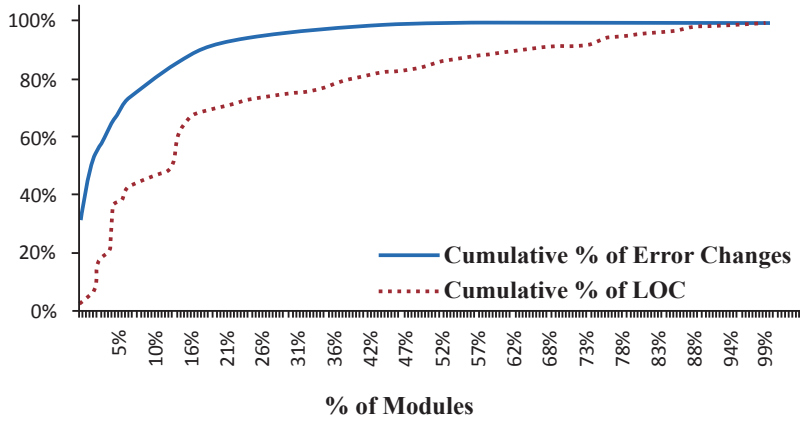

Fig. (6). Cumulative percentage of error changes and LOC versus the accumulative percentage of modules.

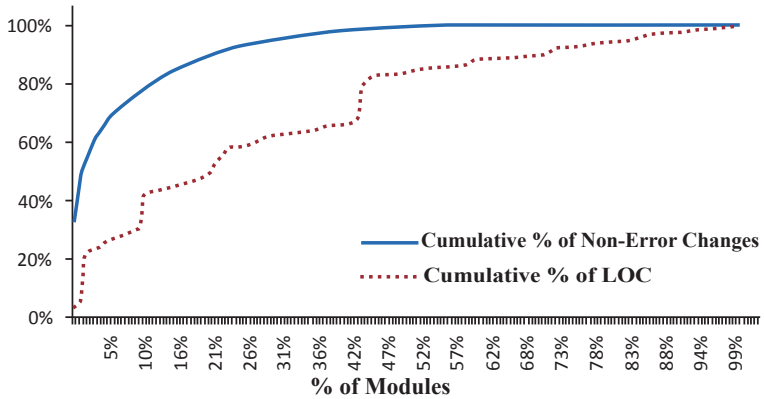

Fig. (7). Cumulative percentage of non-error changes and LOC versus the accumulative percentage of modules.

Table 12. Distribution of Accumulative Percentage of Changes and LOC in Files

# of Files	% of Files	# of Changes	% of Changes	LOC	% of LOC
1696	5.00	30149	60.48	649563	17.18
3391	10.00	38443	77.11	937014	24.79
3871	11.42	39883	80.00	1000427	26.46
5086	15.00	42911	86.08	1216662	32.19
6781	20.00	45820	91.91	1411207	37.33
8476	25.00	47515	95.31	1538897	40.71
10172	30.00	49211	98.71	1736287	45.93
10814	31.90	49853	100.00	1796720	47.53

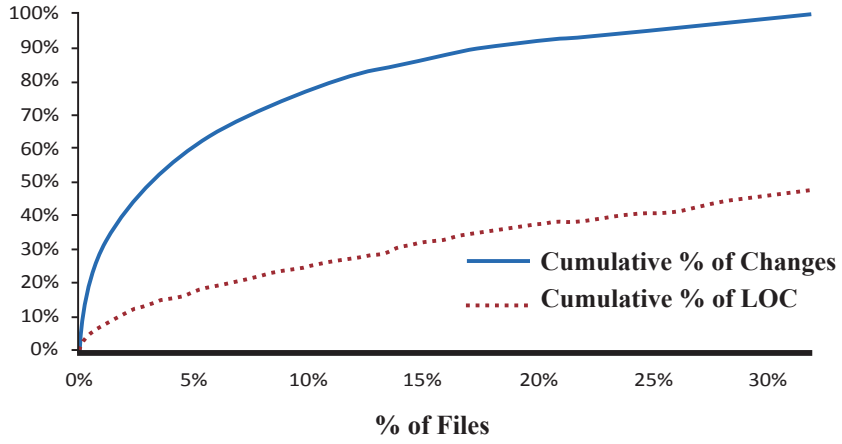

Fig. (8). Accumulative percentage of changes and LOC versus the accumulative percentage of files.

We sorted the modules in the descending order with respect to the number of changes. Fig. (**5**) shows the plot with the accumulative percentage of changes and LOC on the y-axis with respect to the accumulative percentage of modules on the x-axis. Visually, we see that the two curves were fairly close and their trends were similar. Therefore, we accepted Hypothesis #5 for modules.

When changes were examined in two groups: non-error changes and error changes, the phenomenon that we observed for total changes disappeared. Table **10** and **11** summarize the module LOC data for the error changes and non-error changes. Fig. (**6** and **7**) are the plots of the data in Table **10** and **11**. In both Fig. (**6** and **7**), the distance between the two curves was big and the trends between them were not similar. For error changes and non-error changes, we observed different behavior from that of total changes in terms of the accumulative percentage of modules and accumulative percentage of LOC with respect to the accumulative percentage of the changes.

5.6. LOC for Files

Table **12** summarizes the LOC data for files. We sorted the files and LOC for the files in the descending order with respect to the number of changes. Fig. (**6**) plotted the accumulative percentage of changes and LOC on the y-axis with respect to the accumulative percentage of files on the x-axis. We plotted the graph for 10,814 files (about 32% of the total), because these files contained 100% of the changes.

We see in Fig. (**8**) that the two curves were not close and the trends were not similar. The gap between the blue and red lines was too big to accept Hypothesis #5 at the file level.

For Hypothesis #5, we had the mixed results: At the module level, we accepted it only for total changes, not for error and non-error changes. At the file level, we rejected it at the total change level. Once again, we noticed the different behavior between modules and files for the total changes. More discussion on the mixed results is presented in the Discussion of the Findings section.

The conclusion on modules for the total changes seems to contradict the conclusion from that of Hypothesis #4 test. A more detailed discussion on this issue is in the Discussion of the Findings section.

6. DISCUSSION AND CONCLUSION

Finding the power-law distribution, also known as the Pareto principle, as a common phenomenon in the change history of the large object-oriented system in its post-release evolution phase was not surprising. What was unexpected is that the power-law distribution was sporadic at the file level, in contrast to the finding at the module level. This finding by and large agrees with previous research results from [8-10].

Another unexpected finding was the difference between module and file in the relationship between size and the total

changes. We found that the few modules that had the majority of changes (> 80%) contained the majority of code size (>70%). However, the same cannot be said for the files.

These expected and unexpected findings suggest that in large object-oriented systems that are in their post-release evolution phase, the change behavior of the software artifacts is different for modules and files, and is likely to be more predictable at the module level than at the file level.

We found no correlation between the artifact size and changes. This finding goes against the majority opinion that the two are correlated [5, 9, 10, 12, 19, 20]. On the surface, it seems to contradict our own finding that modules, which had the most code size contained the majority of changes. For correlation to exist between two groups of data they must be in synchronization with their increasing and decreasing trends, that is, when one group goes up (or down) in value, the other group must follow suit. This trend apparently did not exist in the data that we analyzed at either the module or file level. However, if we ignore the synchronization aspect of the trend, we did find that, at the module level, the total changes concentrated in a few modules that had the most code size. This finding suggests that the evolutionary changes in large object-oriented systems may not be sensitive to correlation.

Emam and colleagues found the confounding factor of class size in using metrics to predict various factors in software systems [12]. Since most Java files that we studied contained only one class, we considered the file and class to be the same artifact. Our findings showed mixed results. On one hand, at the module level, the total changes concentrated in a few modules that made up the bulk of the system size, although correlation between the two did not exist. On the other hand, the same cannot be said for error changes or non-error changes at the module level; and no such phenomenon was observed for files for any kind of changes. The practical implication of our research findings is that the artifact size alone will unlikely to be an effective predictor for change-prone artifacts for files/classes.

We want to emphasize that all the suggestions and conclusions from our research apply to large object-oriented systems that are in their post-release evolution phase. Large systems that are in their development phase may exhibit different patterns in changes and in the relationship between artifact size and changes.

We have analyzed the maintenance change history of the Eclipse Project that consists of 220 modules (Java packages), 33904 files, 3780201 lines of code, and 49853 changes. We investigated two levels of software artifacts: module and file. At the module level, we found that the power-law distribution was a common phenomenon in total changes, error changes, non-error changes, and for all but one change categories. At the file level, the power-law distribution existed in total changes, error changes, non-error changes; however, the majority of the change categories (about two third) did not show power-law distribution.

For the relationship between artifact size and changes, we found that at the module level, a few modules that had the majority of the total changes accounted for the majority of the code size; however, this phenomenon did not exist for error changes and non-error changes. At the file level, the few files that accounted for the majority of total changes did not account for the majority of code size. We did not find any correlation between artifact size and any kind of changes at either the module or file level. Our findings cast doubt that artifact size alone can predict the change probability for individual files and modules.

Our research findings suggest that in large object-oriented systems, change-proneness prediction will likely be effective at module level for large object-oriented systems that are in their post-release evolution phase.

7. VALIDITY THREATS

There are some limitations in our research. We summarize these limitations as the internal and external threats.

Internal Threats: We collected the change data from the Eclipse Project's public repositories: CVS and Bugzilla. Any changes that were not logged in the two sources were not considered in the study. We make no claim about the accuracy of the data logged in these two sources. The metrics data were collected by using RSM–a commercial metrics tool, - We do not make any claims about the accuracy of the tool, but we believe that the tool collected the data consistently.

The manual classification process is subjective. We classified the changes into 19 categories by using our judgment. In the case where a change might be placed in multiple categories, we used our best judgment to place the change in the most suitable category.

External Threats: The Eclipse Project is a mature industrial software tool that has been in the post-release evolution process for more than a decade. The conclusions from this research are applicable to similar systems: large object-oriented systems that have been released and are in their post-release evolution phase. We do not suggest generalizing our research results to systems that are under development, because we believe that systems under development exhibit different change patterns and different relationship between changes and module/file.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflicts of interest.

ACKNOWLEDGEMENT

Declared none.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*, Addison Wesley: USA, 1999.

[2] V.R. Basili, and B.T. Perricone, "Softw Errors and Complexity: An Empirical Investigation," *Commun. ACM,* vol. 27, No. 1, pp. 42-52, 1984.

[3] V. Y. Shen, T. Yu, S. M. Thebaut, and L. R. Paulsen, "Identlfylng error-prone software: an empirical study," *IEEE Trans. Softw., Eng.*, pp. 317-324, 1985.

[4] C. Withrow, "Bug density and size in ada software," *IEEE Softw.*, vol. 7,1, pp. 26-30, Jan/Feb. 1990.

[5] K.H. Moller, and D.J. Paulish, "An empirical investigation of software fault distribution," *Proc., First Int'l Softw. Metrics Symp.* (Metrics '93), pp. 82-90, 1993

[6] N. Ohlsson, H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Softw. Eng*, vol. 22, no. 12, pp. 886-894, 1996.

[7] L. Hatton, "Reexamining the fault density component size connection," *IEEE Softw*., pp. 89-97, 1997.

[8] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE TSE*, vol. 26, no. 8, 2000.

[9] T. J. Ostrand, E.J. Weyuker, and R. M. Bell, "Where the bugs are," In: *Proc. ACM/International Symposium of Software Testing and Analysis*, Boston: USA, pp. 86-96, July 2004.

[10] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distribution of complex software systems," *IEEE Trans. Softw. Eng*., vol. 33, no. 5, pp. 273-286, 2007.

[11] H. Zhang, "On the distribution of faults," *IEEE Trans. Softw. Eng.,* vol 32, no. 2, pp. 301-302, 2008.

[12] K. El Emam, S. Benlarbi, and S.N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Trans. Softw. Eng*., vol. 27, no.7, pp.630-650.

[13] IEEE standard, "IEEE standard classification for software anomalies," *IEEE Std* 1044-2009 (Revision of IEEE Std 1044-1993), pp. C1-15, 2010.

[14] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou and C. Zhai, "Have things changed now? – an empirical study of bug characteristics in modern open source software," In: *ACM/Architectural Support for Programming Languages and Operating Systems*, San Jose: California, pp. 25-33, 2006.

[15] D. Lo, H. Cheng, J. Han, S. C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," In: *Proc. ACM/International conference on knowledge discovery and data mining*, Paris: France, 2009.

[16] K. Pan, S. Kim, and E.J. Whitehead, "Bug Classification Using Program Slicing Metrics," In: *IEEE/International Workshop on Source Code Analysis and Manipulation*, Philadelphia: USA, pp. 31-42, 2006.

[17] G. Soares, R. Gheyi, T. Massoni, M. Corn´elio, and D. Cavalcanti, "Generating unit tests for checking refactoring safety," *SBLP*, pp. 159-172, 2009.

[18] C. Görg, and P. Weißgerber, "Error detection by refactoring reconstruction," *the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri,* ACM: USA, 2005.

[19] Z. Qin, "Naive Bayes Classification Given Probability Estimation Trees, In: "*Fifth International Conference on Machine Learning and Applications* (ICMLA), Orlando: FL, pp. 34-42, 2006.

[20] J. Thorstern, "A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization," In: *Fourteenth International Conference on Machine Learning* (ICML), Nashvlle, T.ennessee: USA, pp.143-151, 1997.

[21] *CVS and Eclipse*, Available at: http://www.grape-info.com/doc/win2000srv/filemanagement/cvs.html

[22] Boris Beizer, "*Software Testing Techniques*", 2nd ed. The Coriolis Group: USA, 1990.

[23] P. Wolfgang: "*Text Classification Tools Version 0.11*". Available on: http://www.cis.temple.edu/~wolfgang/Text%20Classification%20Tools.pdf [last accessed 15th October, 2012].

[24] A. G. Bluman, "*Elementary Statistics: A Step by Step Approach"*, 2nd ed. Wm. C. Brown Publishers: Dubuque, Iowa: USA, 1995.

[25] Potanin, J. Noble, M. R. Frean, and R. Biddle., "Scale-free geometry in object-oriented programs," *ACM Commun.*, vol. 48, no. 5, 2005.

[26] R. Wheeldon and S. Counsell, "Power law distribution in class relationships," In: *IEEE/International Workshop on Source Code Analysis and Manipulation*, Computer Soc. Press: Los Alamitos, pp. 45-54, 2003.

[27] L. Hatton, "Power-Law Distributions of Component Size in General Software Systems," In: *IEEE Trans Softw. Eng.*, vol. 35, no. 4, pp. 566-572, 2009.

[28] J.M. Juran, F.M. Gryna, Jr., and F.M. Bingham, "*Quality Control Handbook*", 3rd ed, McGraw Hill: New York, 1979.

[29] I.Timea, and D. Barbara, "The vital few and trivial many: An empirical analysis of the pareto distribution of defects," Proc. Lecture Notes in Informatics, pp.151-162, Nov, 2010.

[30] Matlab Official Website, Available at: http://www.mathworks.com/products/matlab/

[31] Aaron, S. R. Cosma, and M.E.J. Newman, "Power-Law Distribution in Empirical Data," *Soc. Ind. Appl. Math.,* vol. 51, no. 4, 2009.

Received: February 02, 2013 Revised: February 12, 2012 Accepted: February 16, 2012