MINING EFFORT DATA FROM THE OSS REPOSITORY OF DEVELOPER'S BUG FIX ACTIVITY

Syed Nadeem Ahsan[1], Muhammad Tanvir Afzal[2], Safdar Zaman[1], Christian Gütel[2,3], Franz Wotawa[1]

[1] Institute for Software Technology (IST)
Graz University of Technology, Inffeldgasse 16b/II 8010, Graz, Austria
{sahsan, szaman, wotawa}@ist.tugraz.at

[2]Centre for Distributed and Semantic Computing (CDSC)
Mohammad Ali Jinnah University, Islamabad, Pakistan
tanvir.afzal@jinnah.edu.pk

[3]School of Information Systems (SIS)
Curtin University of Technology, Perth, Western Australia

Abstract – **During the evolution of any software, efforts are made to fix bugs or to add new features in software. In software engineering, previous history of effort data is required to build an effort estimation model, which estimates the cost and complexity of any software. Therefore, the role of effort data is indispensable to build state-of-the-art effort estimation models. Most of the Open Source Software does not maintain any effort related information. Consequently there is no state-of-the-art effort estimation model for Open Source Software, whereas most of the existing effort models are for commercial software. In this paper we present an approach to build an effort estimation model for Open Source Software. For this purpose we suggest to mine effort data from the history of the developer's bug fix activities. Our approach determines the actual time spend to fix a bug, and considers it as an estimated effort. Initially, we use the developer's bug-fix-activity data to construct the developer's activity log-book. The log-book is used to store the actual time elapsed to fix a bug. Subsequently, the log-book information is used to mine the bug fix effort data. Furthermore, the developer's bug fix activity data is used to define three different measures for the developer's contribution or expertise level. Finally, we used the bug-fix-activity data to visualize the developer's collaborations and the involved source files. In order to perform an experiment we selected the Mozilla open source project and downloaded 93,607 bug reports from the Mozilla project bug tracking system i.e., Bugzilla. We also downloaded the available CVS-log data from the Mozilla project repository. In this study we reveal that in case of Mozilla only 4.9% developers have been involved in fixing 71.5% of the reported bugs.**

Keywords: **Software repository, mining effort data, estimation models, developer expertise and open source software.**

1. INTRODUCTION

In software engineering the cost of any software product is related to the estimated effort that is spent for building the software product. In case of closed and commercial software cost estimation is a continuing activity that initiates at the proposal stage and continues throughout the lifetime of a project, whereas in case of Open Source Software (OSS) the source code is distributed without any cost or limitations. In case of OSS projects effort data is neither stored nor maintained. Therefore, very few attempts have been made to develop an effort estimation model for OSS (Yu, 2006) (Koch, 2008) (Weiss et al., 2007). Most of the research on effort estimation models are related to closed software system (Albrecht and Gaffney, 1983) (Boehm et al., 1995) (Boetticher, 2001). However, the recent trend in software engineering shows that even commercial organization like IBM and SUN have developed OSS products, or have made the code of their proprietary product publically available. The consequences of this trend may impact various software engineering methodologies and project management activities. Especially planning and delivery for OSS will be one of the greatest challenges (Asundi, 2005). An effort estimation model can handle the management issues of OSS such as planning and cost estimation.

During the evolution of any large software system, developers are frequently changing the source code lines of program files. The changes in source code are introduced to enhance the product, i.e., to introduce new features, or to fix detected faults. The changes in the source code are usually stored in version control systems from which effort estimation systems obtain the data that allows for resolving product costs and release management issues. In addition effort estimation systems provide an initial knowledge of the product complexity. Another important feature of accurate effort estimation systems is the reliable and precise scheduling of product release dates. Accurate scheduling of a product release dates always increase the demand of the product and reduce the costs of the project. Despite the discussed importance of effort estimation systems the research community has ignored the area of developing such effort estimation systems for OSS. Consequently, there is lack of theory and methodology for OSS effort estimation (Layman et al., 2008). Most of the OSS projects do not store and maintain effort data. One reason might be the complex development process of OSS. In the following paragraph, we discuss this development process in more detail.

The development process of OSS is totally different to the development process of closed source software. Eric S. Raymond (Raymond, 1999) first described this difference. According to Raymond the conventional closed source development is like building a cathedral: central planning, tight organization and one development process used from the start of the project until its finalization. In contrast the progressive open source development is more like a great babbling bazaar of differing agendas and approaches from which a coherent and stable system seemingly emerge only by a succession of miracles (Raymond, 1999). In case of OSS thousands of projects are currently running. Each project is the product of a community comprising of a few dozens or even hundreds of geographically distributed computer programmers. These programmers voluntarily invest their time and skills for developing the software. They often only work at their personal discretion rather than being assigned and scheduled for some particular tasks (Scacchi, 2005). Within the OSS community volunteer programmers are normally called *contributors*, whereas *developers* are those computer programmers, who have rights to commit changes in the version control system. Developers are therefore the core programmers within OSS project development. There are usually only a small number of developers compared to the number of contributors. Contributors and developers work hand in hand in order to develop OSS. Contributors submit any solution for bug fixes or new features in the form of patches to developers, who validate the submitted patches and commit changes in the version control system. Developers and contributors discuss engineering activities, bugs, and other issues using emails. Because of missing effort data in OSS and the mentioned interaction between contributors and developers there are hurdles to build an effort estimation model for OSS.

In this paper, we present an approach to build an effort estimation model for OSS. For this purpose we construct a developer activity log-book to manage the development activities of developers and contributors. Furthermore, we describe and elaborate several applications of bug fix effort data. Our work is an attempt to extend our previous research (Ahsan et al., 2009) where we presented the idea to mine effort data from software repositories to build a software effort estimation model. In contrast to our previous work we further modified our approach and present an extended and enhanced version of effort model as well as discuss several applications of bug fix effort data in the context of OSS development.

We employ the idea of mining effort data to answer five research questions: 1) how can we mine effort data from the history of developer's bug-fix-activity? 2) How can we build an effort estimation model using mined effort data? 3) How can we use the bug-fix-activity data to construct the developer's activity log-book? 4) How can we use the bug-fix-activity data to estimate developer's contribution measures? And 5) how can we use the bug-fix-activity data to visualize developer's collaborations?

In order to tackle these 5 research questions our paper comprises the following contributions:

1 We devise a novel technique to mine the effort data from software repository.

2 We propose a method for building a developer's log-book that maintains the developer's bug fix activity records.

3 The log-book is used to obtain the actual time spend by each developer to fix a bug. We also show how to discover the expertise level of developers from their bug fix activity data. Moreover, we introduce a way for visualizing the different aspects of the collaboration among the developers.

4 We present empirical results obtained when applying our method for effort estimation on the Mozilla project. In order to perform the experiments we downloaded program file bug-fix-activity data from Mozilla's Bugzilla repository. Bugzilla is a bug tracking system and very commonly used in OSS development. The Bugzilla project data is freely available on their project website (http://bugzilla.mozilla.org). To obtain the developers' bug fix activity data, we first obtained a list of bugs reported in Mozilla project during 1999 to 2007. Then, we downloaded the developers' bug fix activity records of all bugs, which were fixed and closed. As a whole we downloaded 93,607 bug reports.

The paper is organized as follows: In Section 2 we discuss related work. In Section 3 we describe our approach to mine the effort data from software repository. In Section 4 we discuss the developer's activity log-book and the various applications of bug fix effort data in OSS. Finally, in Section 5 and 6 we discuss the existing threats to validity and conclude the paper with some discussions on future work.

2. RELATED RESEARCH WORK

To the best of our knowledge very few attempts have been made to mine effort data from software repository in order to build effort estimation models for OSS. We found that JIRA is the only available bug tracking system[13] that provides the provisions to store and maintain effort information (Weiss et al., 2007). Most of the OSS system like Mozilla, Eclipse and Gnome used Bugzilla as their bug tracking system, which does not allow for maintaining effort information. In the following paragraph, we discuss related work on attempting to build effort estimation or prediction models for OSS. Moreover, in the last paragraph of this section we discuss research work that is related to an approach for finding effort data for OSS.

Weiss et al. (2007) used existing issue tracking system for the JBoss project. In the paper the authors used a framework to identify the similar earlier issue report, and used the average time as a prediction for the effort. For this purpose Weiss et al. used the nearest neighbor approach. Lucas D. Panjer (Panjer, 2007) used data mining tools for predicting the time to fix a bug. Panjer used the Eclipse bug database for evaluating his approach. The proposed model is able to correctly predict 34.9% of the bugs into a discrete log scale lifetime class. Sunghun and Whitehead (Kim and Whitehead, 2006) used ArguUML and PostgreSQL project repository data in their paper. They processed the data to find out the 'bug introduced' and 'bug fixed' time. They compared the total number of bug fixed days with the total number of bugs. Jai Asundi (2005) discussed the importance of effort estimation models for OSS. The author identified that the current effort models are inadequate. Consequently, there is a need for new effort models in the context of OSS development. In this regard the author outlined some issues that should be taken into consideration when developing an effort estimation model for OSS. In particular Asundi suggested an *Activity Based Cost* type model for effort estimation.

Liguo Yu (2006) analyzed an evolutionary data set of 121 revisions of the Linux operating system to develop an effort estimation model for OSS. Since Linux does not maintain effort data, the author first performed an experiment on the NASA SEL database, which is a closed software system and maintained actual effort data. From this experiment Yu identified some measures that can be used indirectly to represent maintenance effort. In the next step, the same measures were employed as effort. The author developed two regression based estimation models for the Linux project. Our work is partially related to Yu's work. But we use our own heuristic approach to obtain the effort data for the Mozilla project. Some other research related to developers contribution measures and visualizations of developer contributions are mentioned in (Schuler and Zimmermann, 2008), (Gousios et al., 2008) and (Lanza and Ducasse, 2002).

3. MINING BUG FIX EFFORT DATA TO CONSTRUCT AN EFFORT ESTIMATION MODEL

In order to meet the objectives of this paper, we first describe the use of software repositories for storing the developer's activity data in this section. Furthermore, we discuss the role of developers and contributors in fixing a bug. Afterwards, we describe our approach and the used techniques for mining effort data for OSS from software repositories.

3.1 OSS Repository (CVS and Bugzilla)

Software repositories in most cases comprise a version control system and a bug tracking system. CVS and Subversion are the most popular and commonly used version control systems. The main functions of a version control system are to store and reconstruct past and current versions of program source files. As a by-product version control system also capture a large amount of contextual information of each change in the program source files (Ball, 1997). The program file log data, which is stored in a version control systems comprises the complete change history of each source file like the revision date, the developer's name, the revision number, comments, and other information of interest. Moreover, in OSS development the most abundant, common and reliable sources for failure information are bug databases. The main function of bug database is to store the history of all the faults that occurred during the software life-time (Schröter, 2006). Hence, bug tracking

[13] http://www.atlassian.com/software/jira/

systems and version controls systems are the two main sources that maintain the developers and contributors development activities. An example of the Bugzilla bug tracking system's interface and the Bugzilla bug life cycle are shown in Figure 1(a) and Figure 1(b) respectively. Because of the stored information in version control systems and bug tracking systems both are used in order to obtain the necessary information for an effort estimation system.

3.2 Role of Developers and Contributors in OSS

The main development difficulties in OSS systems are the managing and monitoring of development activities. In OSS development environments thousands of software contributors provide manpower for developing the OSS system. Usually, the contributors are spread throughout the world, spend tremendous amounts of time and effort in writing and debugging software, and most often with no direct monetary rewards (Joseph and Brian, 2001). In OSS development developers are those who have the right to commit the changes in the version control system whereas the software contributors have no such rights. Instead the contributors are allowed to submit the solution in the form of patches to the developers. Usually there is only a small number of developers compared to the number of contributors (Alonso et al., 2008). When a bug is reported, normally developers (who are working as software quality assurance personals) validate the reported bug, and assigned it to any appropriate contributor. Alternatively, sometimes contributors voluntarily take a challenge to fix new reported bugs. After resolving the bug, contributors submit patches providing a solution. These patches are then validated and committed by developers into a version control system. These bug-fix-activities are stored into a version control system and a bug tracking system. Because of this process of bug fixing one can argue that a person committing a change is not necessarily an expert. It reveals that in case of OSS measuring effort and expertise is not trivial. In the following section, we describe the data extraction process and describe how we used the extracted bug-fix-activity data to construct an effort estimation model.

3.3 Data Extraction from Software Repository

To perform the experiment we selected the Mozilla open source project as our source for the version control system and the bug database. For this purpose, we downloaded developers' bug fix activity data where the bug activities have been reported and fixed between Nov-1999 and Dec-2007. We also obtained the list of bug ids from MSR 2007 data repository (Weiss et al., 2007). We used the wget tool to extract the data from the Bugzilla web site. We preprocessed the downloaded html files and extracted the developers' bug fix activity data. Furthermore, we connected to the Mozilla CVS server and downloaded CVS log data and source file revisions. Unfortunately, in case of OSS bug databases are not directly linked with a version control system. Therefore, there is no direct mapping between the fixed bug reports and the name of the fixed/updated source files. However, in recent years, a number of techniques have been developed to relate bug reports to fixes in source files. To establish a link between a bug report and the list of fixed source files, we used an approach similar to M. Fischer's approach (Fischer et al., 2003). Once, we obtained all the required data, we used this data to build an effort estimation model as explained in the next subsection in detail.

3.4 Construction of Effort Estimation Model for OSS

In this subsection we focus on the computation of the time that is spent to fix a bug. For this purpose we first explore the bug life cycle of Bugzilla in more detail. The complete bug life cycle of Bugzilla is shown in Figure 1(b). According to the bug life cycle the quality assurance (QA) person assigns any new bug to a developer or contributor for providing a solution. Hence, the real effort for fixing a bug starts when a bug is moved from NEW to the ASSIGNED state. The bug is solved when the developer or contributor provides a solution and moves the status to RESOLVED. The duration between ASSIGNED and RESOLVED is the actual period where effort is spent on source code changes to fix the bug. We assume that this time period is equivalent to the actual bug fix time and, therefore, the only time to be considered as bug fix effort. Note that in some cases a QA person reassigns the same bug to another developer or in some cases the previously assigned developer assigns the bug to some other developer, which makes the computation of the overall effort even more difficult. Since no information regarding the distribution of effort among the different developers is available, we assume that all developers contribute. Thus we calculate the sum of all time periods for each developer or contributor to come up with a single total bug fix effort.

For a single developer, we compute the effort required to provide a solution to a bug report as follows: We start with the bug reports assigned to the developer. Subsequently, we compute the effort assigned to a bug report for each month of the year using the given dates for ASSIGNED and RESOLVED. The time span between ASSIGNED and RESOLVED cannot be directly used to compute the effort. The reason is that a developer works on several bug reports in parallel but it is impossible to spend more than all days of a month in working. Hence, the time spent has to be multiplied by a factor. This factor takes into account the limited number of days available for working within a particular month. Therefore, we have to multiply the duration of bug fix with a common multiplication factor (M_k). We obtain the multiplication factor by dividing the total

working days of a month (T_w) with the sum of all the assigned working days for all the assigned bugs, i.e., $M_k = T_w / \sum Days$,

$$E_i = \sum (Bug\ fix\ duration\ for\ bug_i)_k \times M_k. \quad (1)$$

Where, E_i is the estimated bug fix effort of bug_i, which is the sum of the bug fix duration that is spent in k months. If a bug is fixed in more than one month, then the bug fix duration for the first month is obtained by subtracting the bug assigned day from the last day of the month, and for the last month, the bug fix duration is obtained by subtracting the first day of the month from the bug resolved day. We consider the whole days of a month as bug fix duration for all of the intermediate months. We multiply each month's bug fix duration with the respective multiplication factor. Finally, we add all the obtained values to get an estimated time spent to fix a bug, which we assume as the estimated effort value. But, during this experiment we found that the bug severity level affects a bug fix time. Therefore, we further improve our effort estimation model by taking into account the bug severity level. The bug severity level is defined by a user at the time of submitting a new bug to the bug database. On the basis of severity level, a developer assigns a priority level to each assigned bug report. In our experiment, we found that high severity and high priority bugs are fixed in less time compared to low severity and low priority bugs. The average bug fix time related to different severity level is listed in Table 1. From the data of Table 1, one may conclude that the bug fix time is dependent on the severity level. In the following paragraph we elaborate the relation between bug severity level and bug priority level.

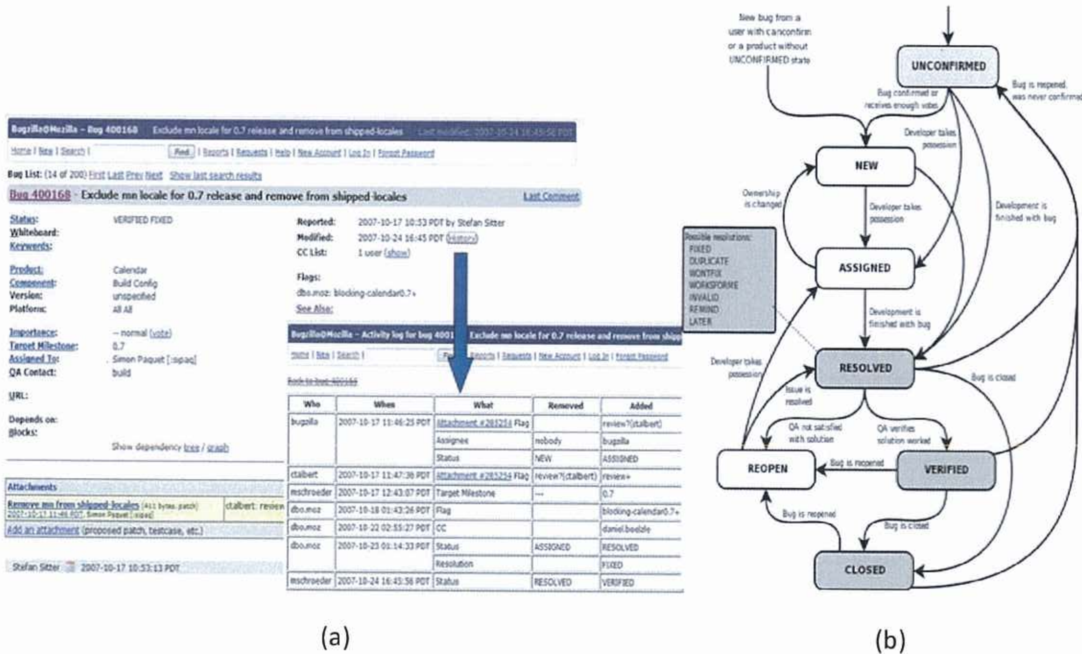


Figure 1. (a) A GUI interface of Bugzilla bug tracking system. The upper part of the base window displays the complete information of bug: 400168 i.e., bug title, status product etc., while the bottom part of the base window displays the details about the patch attachments, and communication between developers and contributors. The popup window gives the complete history of bug. Whereas, each row contains entries of developer bug fix activities. (b) A complete bug life cycle of Bugzilla, each rectangle represents a unique status of a bug. The time spent between ASSIGNED and RESOLVED is the actual time spent to fix a bug.

In case of OSS, developers or contributors do not treat bugs equally. Rather bugs are treated on the basis of their severity level. In case of the Mozilla project, bug reports have seven different severity levels, namely critical, blocker, major, normal, trivial, minor and enhancement. At the time when a user reports a new bug, an appropriate severity level is associated with the bug report. Before fixing any bug report, developers, first assign a priority level to the bug report. In case of Mozilla, there are five different type of priority level i.e., P1, P2, P3, P4 and P5 (Herraiz et al., 2008). During the experiment, we found that it is not necessary for a developer to assign any fix priority to any severity level. However, it should be mentioned that a developer not necessarily treats a bug of critical or blocker severity level at a high priority i.e., P1 (see Table 1). Table 1 shows that developer assigned different priority levels to the same severity level. Hence, there is no one-to-one mapping between severity and priority levels. Figure 2 shows the frequency distribution of different priority levels

associated to each severity level. A plot comparing the bugs of different severity level and their bug fix time is shown in Figure 3, which depicts that less time is spent to fix high severity level bugs.

In order to find the impact of bug severity level on bug fix effort, we determine the correlation between the bug severity levels and a set of source file metrics. We use those set of source file metrics, which measure the degree of complexity in a source files. The results are shown in Table 2, which shows that there is no significant correlation between the bug severity levels and the source file metrics. Therefore, we may conclude that bug severity levels are not related to the complexity of source files. However, developers spend less time to fix high severity level bugs, not because they are less complex and therefore, need less time or effort to fix. But in fact high severity level bugs should be fixed in short time. Similarly, less severity bugs, which take a longer time to be fixed, are not necessarily less complex. In order to treat all of the bugs equally, and to nullify the impact of a biased attitude of developers from the bug fixing time, we divided the estimated effort, obtained in Equation 1 by the time scaling factor *Severity Factor* (SF_j). Where, SFj has three values i.e., $SF_1 = 1$ for critical, blocker and major severity level, $SF_2 = 2$ for normal severity level, and $SF_3 = 3$ for trivial, minor and enhancement severity level (see Table 1).

$$E_i = [\sum (Bug\ fix\ duration\ for\ bug_i)_k \times M_k] / SF_{ij} \quad (2)$$

To understand, how we estimate the bug fix effort from bug reports, consider an example, in which a developer fixed four bugs in February, 2008. All of the bugs were critical i.e., $SF_{ij} = 1$. The example data is shown in Figure. 4. The gray bar represents the durations in which the developer fixed those bugs. We obtain these durations from bug reports by subtracting the bug assigned date from the bug resolved date. In this example, we assume that during some days of the month, the developer worked in parallel on multiple bugs, now we apply our model to estimate the effort value.

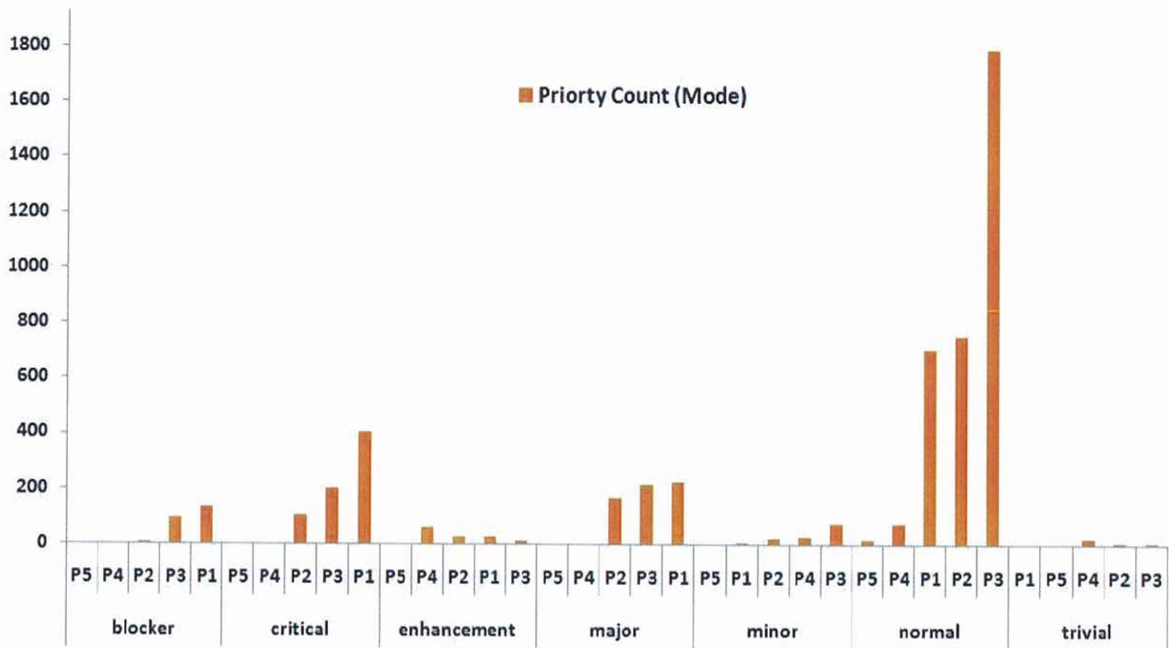


Figure 2. The frequency distribution of different priority levels i.e., P1, P2, P3, P4 and P5, which are assigned to different bug severity level.

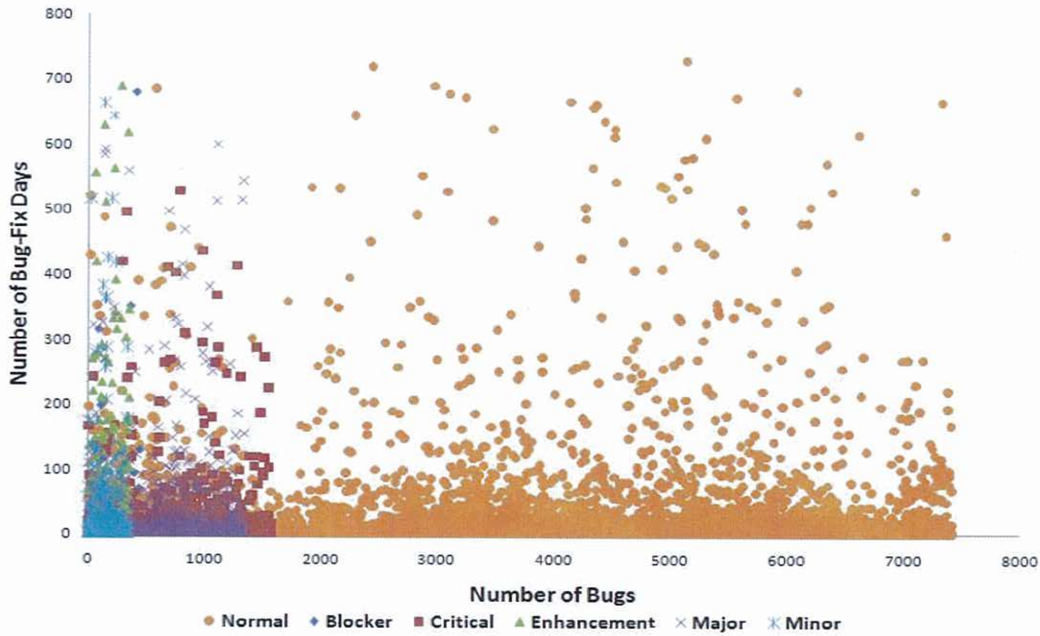


Figure 3. A plot between the bugs of different severity level and the number of days spent to fix them. Each bug is represented by a dot where the colour of the dot represents its severity level.

Developer Name: X										**Month: February, 2008**																		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	Days
									Bug Id= 1, Bug fix duration 06 days																			06
		Bug Id= 2, Bug fix duration 10 days																										10
					Bug Id= 3, Bug fix duration 21 days																							21
														Bug Id= 4, Bug fix duration 10 days														10
		Total working days T_w = 24 days																								∑*Days*		47

Figure 4. An example of developer 'X' bug fix activity. Developer 'X' fixed four critical bugs during February, 2008.

$M_k = T_w / \sum Days$= 24/47 = 0.51 and $E_i = (\sum (Bug\ fix\ duration\ for\ bug_i)_k \times M_k) / SF_{ij}$, Where E_i is the bug fix effort for bug i, and k is the number of months spent to fix a bug. In this example k=1. Therefore, E_1 = (6×0.51)/1 = 3.06. Similarly we estimate the effort for other bugs, E_2=5.1, E_3=10.7, & E_4=5.1. We built a program that automatically performs all of the mentioned steps to obtain the effort value and store it into a database.

Table 1: Classification of bug reports on the basis of bug severity level and developer defined bug priority.

Bug Severity Level	Severity Weight (SW)	Number of Bugs	Average Bug-Fix Time (days)	Standard Dev. Bug-Fix Time	Priority (Assigned by Developer)	Severity Factor
blocker	7	412	11	0.4434	P1,P3	1
critical	6	1413	20	0.2660	P1,P2,P3	1
major	5	1071	103	2.5362	P1	1
normal	4	7005	45	0.6571	P3,P4	2
trivial	3	152	58	2.8388	P4	3
minor	2	314	50	2.5959	P4	3
enhancement	1	326	119	3.1015	P4,P5	3

Table 2: Person and Spearman's Correlation of bug severity level with a set of source file metrics.

Correlation of Bug-Severity	Total Pointers	Total Logical Operator	Total Relational Operator	Total Function Call	Total Assert	Total Arrays	Total Function Declaration
Pearson	0.026	0.031	0.026	0.014	0.031	0.004	0.018
Significance level	0.068	0.038	0.065	0.205	0.036	0.419	0.154
Spearman's	-0.033	0.002	-0.036	0.009	0.035	0.001	0.014
Significance level	0.029	0.450	0.018	0.293	0.022	0.490	0.205

4. APPLICATION OF DEVELOPER'S BUG FIX ACTIVITY DATA IN OSS

In the previous section, we described our approach to extract effort data from the history of the developers' bug-fix-activities, and discussed our effort estimation model. In this section we describe some applications of developer bug-fix-activity data.

4.1 Developer Activity Log Book

In commercial software organization, different tools and techniques are available to monitor and control the developer's monthly development activities (Pressman, 2000). One of the techniques is to store and maintain each developer's activity logs in the developer's log-book. The developer activity log provides past and present development history related to the developer. In OSS there is no such system, which logs the developer or contributors[14] development activities. However, this information can be mined from the developer bug fix activity records stored in bug tracking systems. The developer's log-book contains useful data, which can further be processed intelligently to obtain the following information.

1. Development work done by each developer in any month of a year. This may be used to analyze the developer's performance and the development load on each developer in any month of a year.
2. Time spent by each developer to resolve the assigned development task. This may be used to analyze the developer efficiency and expertise.
3. Total time spent by one or more developers to fix a bug. This may be considered as bug fix efforts in days.

We use the log-book data to calculate the bug fix effort data. The Log-book contains the complete history of each bug and the name of developers whoever have been involved in bug fixing. An example of the automatically generated log-book is shown in Figure 5. Besides its main advantage of providing the effort data, there may be several other advantages of log-books. For example, a log-book may be used for the analysis of developer's activity patterns. It is shown in the pop up window of Figure 5 that the developer *Neil* is more active during the last months of the year 2001 as compared to the initial months of the same year. Similarly, we can use this log-book to analyze the month wise or year wise effort distribution of developers.

For the Mozilla project, we used the developer's activity log-book data to obtain the average bug fix efforts in days. We added all the bug fix days during any year and divided it by the number of bugs fixed in that year. Figure 6(a) shows a comparison between the total number of bugs fixed in a year and the corresponding average bug fix efforts. It is observed from that figure that during the initial years of the project, small effort was required to fix a large number of bugs. However, as the time advances the count of bug fixed per year decreases, while the average effort to fix bugs increases. Figure 6(b) shows the relationship between the bug fix days with the frequency of bug's occurrence in that period. It is observed that for Mozilla project, the frequency of the bugs that needs small effort (actual bug fix days) is much larger as compare to those which need more efforts.

[14] In OSS, both developers and contributors are computer programmers; the main difference is developers have more rights as compared to contributors (see, Section 3.2). Now from this point and onward we use the word developers instead of "developers or contributors".

4.2 Developer's Contributions Measures to Identify Expertise

During the evolution of large software systems, it is essential to identify those individuals who are expert in some specific part of the product. The current approaches for expertise recommender systems are mostly based on variations

Developer Name: neil@httl.net — Log Year: 2001

Bug Id	Date Assigned	Date Resolved	Jan	Feb	Mar	Apr	May	Jun	July	Aug	Sep	Oct	Nov	Dec	Total Days
1697	2001-04-25	2003-07-31	00.00	00.00	00.00	02.31	08.74	06.47	04.43	04.11	03.75	03.31	02.86	02.28	038.26
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	08.74	06.47	04.43	04.11	03.75	03.31	02.86	02.28	113.95
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	00.00	00.00	00.00	00.00	00.00	022.31
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	00.00	00.00	00.00	00.00	00.00	001.47
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	00.00	00.00	00.00	02.03	02.86	007.17
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	08.74	06.47	04.43	01.99	00.00	00.00	00.00	00.00	039.02
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.57	04.11	03.75	03.31	02.86	02.28	016.88
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	04.79	06.47	04.43	04.11	03.75	03.31	02.86	02.28	032.00
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	03.24	04.43	04.11	03.75	03.31	02.86	02.28	023.98
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.86	04.43	04.11	03.75	03.31	02.86	02.28	021.60
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	03.86	04.11	03.75	03.31	02.86	02.28	020.17
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	02.60	03.75	03.31	02.86	02.28	012.46
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	00.00	00.00	02.46	02.86	02.28	007.60
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	00.00	00.00	00.00	01.33	02.28	003.61
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	00.00	00.00	00.00	00.00	02.14	002.14
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	00.00	00.00	00.00	00.10	00.00	000.10
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00.00	00.00	00.00	00.00	00.00	00.00	00.00	01.47	001.47
116196	2001-12-20	2002-09-13	00.00	00.00	00.00	00.00	00.00	00.00	00.00	00.00	00.00	00.00	00.00	00.81	000.81
Total days/month used to fix a bug			31.00	28.00	31.00	30.01	31.01	29.98	26.58	31.00	30.00	28.94	29.20	30.08	365
Number of bugs fixing/month			2	2	2	3	4	6	8	9	8	11	12	14	

Developer Name: neil@httl.net — Log Year 2001

Date Assigned	Date Resolved	Jan–Dec
2001-04-25	2003-07-31	Bug Id: 1697
2000-05-16	2003-04-07	Bug Id: 38367
2000-12-28	2001-02-09	Bug Id: 54175
2001-12-11	2005-03-02	58523
2001-10-12	2004-11-28	Bug Id: 58523
2001-03-27	2001-08-15	Bug Id: 66475
2001-07-27	2003-02-28	Bug Id: 75686
2001-05-14	2003-03-20	Bug Id: 80837
2001-06-15	2003-09-11	Bug Id: 85908
2001-06-26	2003-09-11	Bug Id: 87924
2001-07-04	2003-10-17	Bug Id: 89212
2001-08-29	2005-09-30	Bug Id: 97532
2001-10-08	2002-12-12	Bug Id: 99328
2001-11-16	2003-01-17	Bug Id: 107418
2001-12-02	2002-01-21	110254
2001-11-27	2001-11-28	111606
2001-12-11	2002-01-15	114522
2001-12-20	2002-09-13	116196

Figure 5. An example of developer bug fix activity log-book

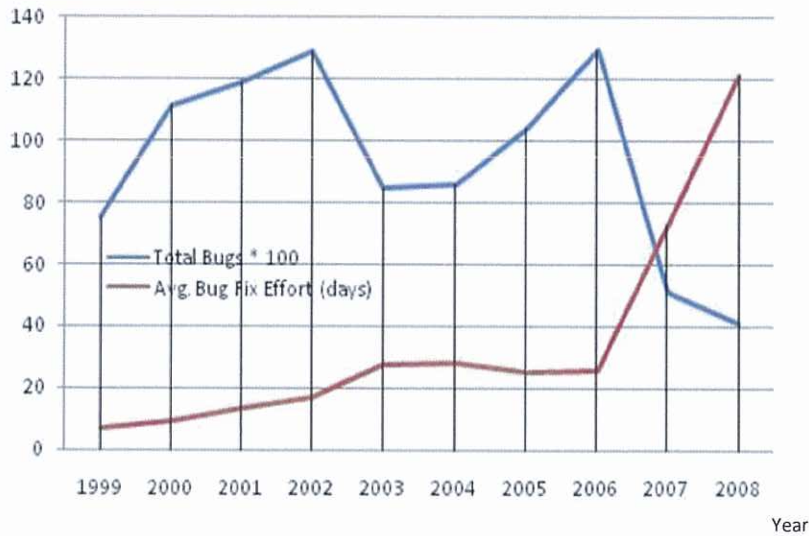


Figure 6 (a). Mozilla project average bug fix effort per bug related to the number of bugs fixed per year.

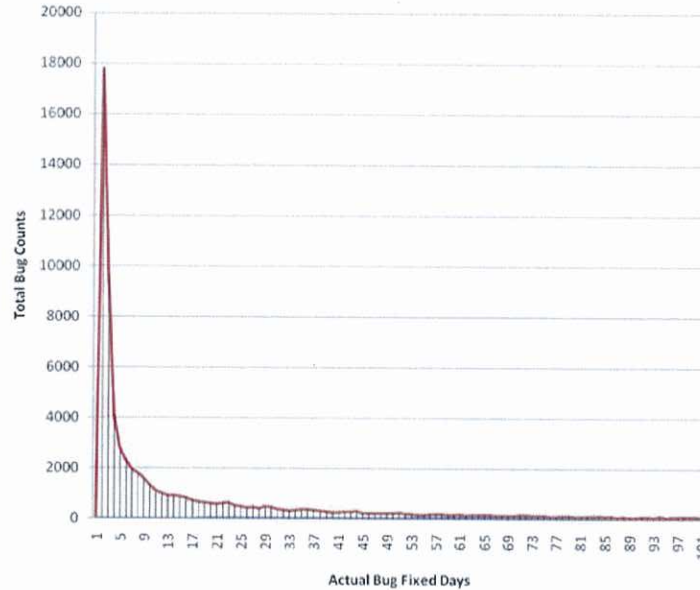


Figure 6(b). Number of bug counts as function of the number of bug fix days.

of the *Line 10 Rules*. According to the *Line 10 Rule,* developers who changed a file most often do have the most expertise for implementation. The name *Line 10 Rule* is come from a version control system that stores the author who did the source file change commit, in line 10 of the commits' log message (Schuler, 2008). In short, developers who commit changes in source files are considered to have expertise in changing those source files. In order to identify the expertise level, we need to measure the developer's contributions during the evolution of software. One can discuss the contribution in different contexts. However, a contribution is commonly related to the developer's productivity or developer's activities (Gousios, 2008). In our experiment, we used the developers' bug fix activity data to measure a set of developers' contribution metrics. These contribution metrics are used to list the name of expert developers.

We propose three different metrics for measuring a developer's contributions. Let DCM stand for Developer's Contribution Measure. The first metric DCM-NBFixed, is the sum of bugs fixed by a developer. The second metric i.e., DCM-WNBFixed is the weighted sum of bugs fixed by a developer. The third metrics DCM-NFFixed is the sum of source files fixed by a developer. The weighted bug fixed is obtained by multiplying each fixed bug with a weighting factor. The weighting factor (SW) represents the severity level of the fixed bug. The severity level was defined in Section 3.3 (see severity weight column of Table 1). The formal representation of developer's contribution metrics are given below,

$$\text{DCM-NBFixed}_i = \sum_j \text{Bug-Fixed-Id}_i(j)$$

$$\text{DCM-WNBFixed}_i = \sum_j \text{Bug-Fixed-Id}_i(j) \times \text{SW}(j)$$

$$\text{DCM-NFFixed}_i = \sum_j \text{Fixed-Source-File-Id}_i(j)$$

Where, i represent developer's id, and j represents bug id. In order to obtain the defined set of contribution measures, we further processed the developer activity log-book, and obtained a metrics value for each developer. We performed this experiment on a sample data set comprising 10,693 bug reports. 410 different developers fixed these bug reports. Figure 7, shows the contribution measures of the top 10 developers. Figure 8(a) depicts the metrics distribution of the top 20 developers, and Figure 8(b) shows the relation between the percentages of bug fixed and the developer who fixed the bugs. It is clear from the figure that only 4.9% developers fixed 71.5% bugs. In our experiment, we found that the most expert developer in Mozilla project is 'bzbarsky@mit.edu'.

Developers	DCM-NBFixed	DCM-WNBFixed	DCM-NFFixed
bzbarsky%mit.edu;	1730	7072	490
roc+%cs.cmu.edu;	883	3650	192
jst%netscape.com;	717	3179	282
karnaze%netscape.com;	716	3312	125
cbiesinger%web.de;	[illegible]	[illegible]	[illegible]
bryner%brianryner.com;			
waterson%netscape.com;			
aaronleventhal%moonset.net;			
timeless%mozdev.org;			

Developer	BugSeverity	SeverityFactor	BugFixed	DCM-WNBFixed
bzbarsky%mit.edu;	blocker	7	16	112
	critical	6	161	966
	major	5	132	660
	normal	4	1256	5024
	minor	3	42	126
	trivial	2	61	122
	enhancement	1	62	62

Figure 7. A list of top 10 Mozilla developers on the basis of the contribution metrics. The pop up window shows the details of the metric DCM-WNBFixed.

4.3 *Visualization of Developer Bug Fix Activity*

Software evolution needs to handle a huge amount of data, which is the major problem and a strong reason for doing research in this field. The huge amount of data comes from analyzing several versions of the same software in parallel. A technique, which may be used to reduce this complexity, is *software visualization* that allows researchers to study multiple aspects of complex problems in parallel (Lanza and Ducasse, 2002). In our experiment, we figured out that the developer's bug fix activity data could be used to visualize the developer's interactions. Therefore, we developed a visual tool using Java API. This tool may be used to explore interesting patterns among the developers and the corresponding source files.

We use the bug fix activity data to visualize the interaction between two or more developers, and the interaction of developers with their assigned source files. The main features of our visualization tool are shown in Figure 9. The upper part of the figure shows the connection between developers. Each circular node represents a developer.

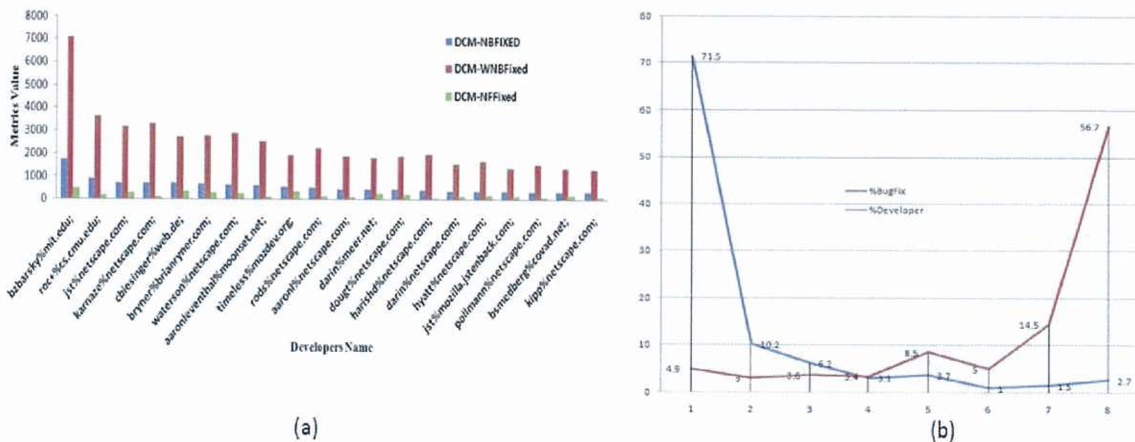


Figure 8. (a) Distribution of five different contribution metrics of the top 20 Mozilla developers. (b) Percentage of bug fixes compared with the percentage of developers fixing them.

The size of the circle represents the developers total bug fix activity. The number inside the circle is the developer's id. The lower part of the figure represents the relationship between source files and developers. Squares represent a source file and circles represent developers. If developers have performed one or more changes in any source file, then there is an edge between the developer and the source file in the graph. The thickness of the edge represents the number of times a developer have changed a source file. The label attached with each link depicts the names of connected developer and source file. The number inside the square is the source file's id. Developers are represented by red circles and the source files are represented by a red squares.

The different size of the circle represents the developer bug fix activity count. From the analysis of Figure 8, we can conclude that in case of Mozilla very few developers have fixed large number of bugs, whereas a majority of the developers have fixed very few bugs. Another interesting finding is that some source files are connected with a large number of developers. This means that multiple developers have changed these source files. Therefore, these types of source files are risky to modify.

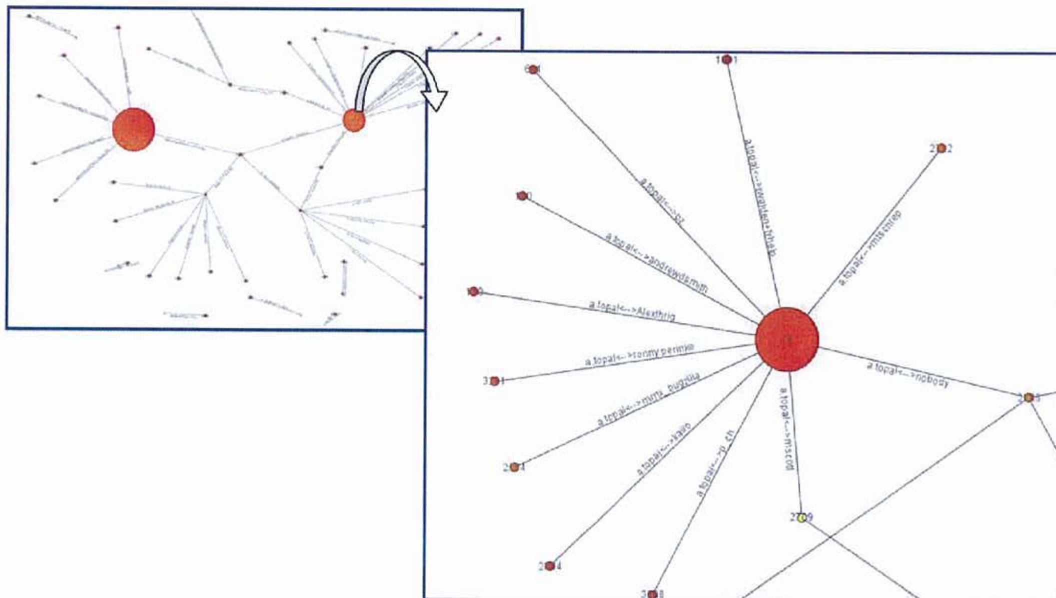
4. THREATS TO VALIDITY

The computation of effort (spent to correct a bug) described in a bug report assumes that the real effort is distributed evenly. This might not be the case generally but since there is no other information available, it is the best we can do. This assumption and some of the other assumptions introduce some errors in the resulting data. But given the huge amount of data available in the repositories, we expect that there is no error inherently present in the process of effort computation. Therefore, there might be a decrease of reliability in the data but there should always be an upper bound.

We also assumed that developers spend the whole assigned period in fixing the bug, but this might not be generally true, because in OSS development only some experienced developers do a full paid job, whereas most of the contributors are volunteers and they are most likely be involved in other jobs during the bug assignment period. Moreover, one cannot completely rule out the existence of any outliers. However, we have removed most of them from our dataset.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented an innovative approach to mine the bug-fix effort data from the history of developer's bug fix activities. This bug fix activity data is extracted from software repositories. We described the different applications of developer's bug fix activity data in OSS. First, we used the extracted data to build an effort estimation model. Subsequently, we devised a method to construct the developers' activity log-book. Furthermore, for measuring the comprehensive expertise level, we defined three metrics. Finally, we used the bug fix activity data to visualize the bug fix interactions among the developers, and also visualize the developer relations with source files. From the analysis of log-book data, we found that in case of Mozilla project only 4.9% of the developers fixed 71.5% of the bugs. We also found that initially little effort was required to fix bugs, whereas as the time advances more effort is required to fix the same number of bugs. In the future, we will further refine the presented work by adding more data from other OSS projects. We will use log-book data, and perform social network analysis for OSS developers. We will further improve our visualization tool to present collaboration among groups of developers.



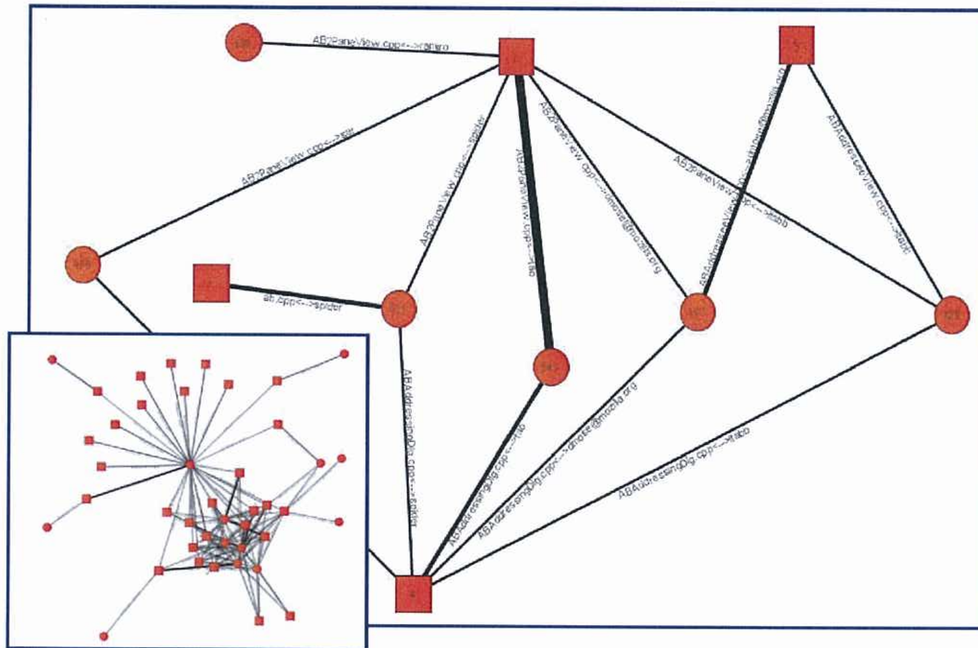


Figure 9. A visualization of the developer's bug fix activity data. The upper part depicts the interaction among developers, whereas the lower depicts the interaction between source files and developers.

ACKNOWLEDGEMENTS

The research work presented in this paper has been partially funded by the Higher Education Commission (HEC), Pakistan and partially conducted within the competence network Softnet Austria (www.soft-net.at) that is funded by Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Witschaftsfröderun-gsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for Innovation and Technology (ZIT).

REFERENCES

Ahsan,S.N., Ferzund, J., Wotawa, F. (2009). Program File Bug Fix Effort Estimation Using Machine Learning Methods for OSS, In proceddings of 21st Software Engineering and Knowledge Engineering (SEKE), pp.129-134, Boston, USA.

Albrecht, A.J., Gaffney, J.E., Jr. (1983). Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, IEEE Transactions on Software Engineer, vol. SE-9, no.6, pp. 639-648, Nov. 1983

Alonso, O., Devanbu, P.T., Gertz M. (2008). Expertise identification and visualization from CVS. In Proceedings of the 2008 international Working Conference on Mining Software Repositories (Leipzig, Germany, May 10 - 11, 2008). MSR '08. ACM, New York, NY, 125-128. DOI= http://doi.acm.org/10.1145/1370750.1370780

Asundi. J. (2005). The need for effort estimation models for open source software projects. SIGSOFT Software Engineering Notes 30, 4 (Jul. 2005), 1-3. DOI= http://doi.acm.org/10.1145/1082983.1083260.

Ball, T., Kim, J.-M.., Porter, A. A. Siy, H. P(1997). If your version control system could talk ... In ICSE '97 Workshop on Process Modeling and Empirical Studies of Software Engineering, May 1997.

Boehm, B. Clark, B Horowitz, E., Westland, C., Madachy, R., Selby, R. (1995). Cost Models for Future Software Life Cycle Process: COCOMO 2, Annals of Software Engineering, 1995.

Boetticher, G. (2001). An Assessment of Metric Contribution in the Construction of a Neural Network-Based Effort Estimator, Second Int. Workshop on Soft Computing Applied to Software Engineering, 2001.

Fischer, M.. Pinzger, M., Gall. H. (2003). Analysing and relating bug report data for feature tracking. In Proceeding of 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria British Columbia, Canada, 2003 IEEE

Gousios, G Kalliamvakou, E, and ,Spinellis, D.(2008). Measuring developer contribution from software repository data. In Proceedings of the 2008 international Working Conference on Mining Software Repositories (Leipzig, Germany, May 10 - 11, 2008). MSR '08. ACM, New York, NY, 129-132. DOI= http://doi.acm.org/10.1145/1370750.1370781.

Hahn, J., Moon, J.Y. and Zhang, C. (2006) Impact of Social Ties on Open Source Project Team Formation. In The Second International Conference on Open Source Systems - OSS 2006, (Como, Italy, 2006).

Herraiz, I., Daniel M. G., Jesus M. G., Gregorio R (2008) Towards a simplification of the bug report form in eclipse. Proceedings of the 2008 international working conference on Mining software repositories, 2008, Leipzig, Germany. http://doi.acm.org/10.1145/1370750.1370786.

Joseph, F., Brian, F. (2001). Understanding Open Source Software Development. Addison Wesley, Pearson Education Book, Dec 2001

Kawin N., Dongsong Z., A Gunes K., Lina Z., Rober N. (2008). An Exploratory Study on the Evolution of OSS Developer Communities. Proceedings of the 41st Hawaii International Conference on System Sciences 2008

Koch, S. (2008). Effort Modeling and Programmer Participation in Open Source Software. Projects, Information. Economics and Policy, 20(4): 345-355.

Lanza, M., Ducasse. S. (2002). Understanding Software Evolution using a Combination of Software Visualization and Software Metrics. In Proceedings of LMO 2002.

Panjer, L.D. (2007) Predicting Eclipse Bug Lifetimes, Mining Software Repositories, ICSE Workshops MSR apos;07. 20-26 May 2007 Page(s):29 – 29.

Pressman. R.S. (2000) Software Engineering: A Practitioner's Approach. McGraw Hill Higher Education; 5th edition. December 1, 2000. ISBN-13 978-0071181822.

Raymond, E.S. (1999) The Cathedral & the Bazaar. O'Reilly Retrieved, 1999.

Scacchi, W. (2005) Socio-Technical Interaction Networks in Free/Open Source Software Development Processes S.T. Acuña and N. Juristo (eds.), Software Process Modeling, pp. 1-27, Springer Science and Business Media Inc., New York, 2005.

Schröter, A., Zimmermann, T., Premraj, R., Zeller, A (2006).. If your bug database could talk. In Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters. 2006.

Schuler, D. Zimmermann, T. (2008). Mining usage expertise from version archives. In Proceedings of the 2008 international Working Conference on Mining Software Repositories (Leipzig, Germany, May 10 - 11, 2008). MSR '08. ACM, New York, NY, 121-124. DOI= http://doi.acm.org/10.1145/1370750.1370779.

Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How Long Will It Take to Fix This Bug?, In Proceedings of the Fourth international Workshop on MSR (20-26 May, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 1.

Yu, L. (2006). Indirectly predicting the maintenance effort of open-source software. Research. Articles, Journal of Software Maintenance and Evolution, (5 Sep, 2006), pages: 311-332.