ALLOCATING TEAMS CONSIDERING EXISTING AND NEW DEVELOPERS FOR BUG ASSIGNMENT

AFRINA KHATUN
Masters of Science in Software Engineering,
Institute of Information Technology, University of Dhaka
Class Roll: MSSE 0402
Registration Number : 2010-812-817

A Thesis
Submitted to the Masters of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

Masters of Science in Software Engineering

Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh

ALLOCATING TEAMS CONSIDERING EXISTING AND NEW DEVELOPERS FOR BUG ASSIGNMENT

AFRINA KHATUN

Approved:

Signature *Date*

Student: Afrina Khatun

Supervisor: Dr. Kazi Muheymin-Us-Sakib

To *Fareda Yasmin*, my mother
who has always been there for me and inspired me

Abstract

Bug assignment is an important activity for faster bug resolution. Existing techniques suggest a bug fixing team using only collaborations on fixed reports. However, if the developer's recent activities are not considered, inactive fixers may have a chance to be included in the team. Ironically, both previous reports and recent commits do not contain any record regarding the newly joined developers. So, to make the situation worst, new developers are never been considered.

For eliminating the allocation of inactive developers, this thesis, first combines the previous and recent activities in individual fixer suggestion. A team allocation technique is then proposed using the concept of combining the expertise and recency of developers. As the new developers may come with a set of expertise, this combination enables the proposed technique to ensure task assignment to both existing and new developers.

To combine the previous and recent activities in fixer suggestion, an Expertise and Recency Based Bug Assignment (ERBA) technique has been proposed. It first processes source code and commits to construct an index connecting the source entities with developers' source activities. Next, it takes fixed bug reports and builds another index, mapping the keywords with developers' fixing expertise. For new reports, it queries the two indexes, and applies tf-idf technique on the query results to calculate a score for ranking the developers. For experimental analysis, ERBA is applied on three open source projects - Eclipse JDT, AspectJ and SWT. The results show that 30.8%, 20.5% and 22.7% actual

developers are suggested at the first position for Eclipse JDT, AspectJ and SWT respectively, which are higher than two existing approaches named ABA-time-tf-idf and TNBA.

The proposed team allocation technique named as TAEN (Team Allocation to Existing and New Developers), first identifies the bug types of the fixed reports using Latent Dirichlet Allocation (LDA) modeling, to create groups of developers who worked on same bug types. A collaboration network is also constructed using the fixed reports, which represents the communication among developers. For new developers, TAEN elicits their preferred bug types by presenting them with most representative terms and bug reports of each type, and adds them to the developers group of the preferred bug type(s). On arrival of a new report, TAEN determines its bug type to fetch the potential developers grouped under this type. It derives the collaborations among the potential developers and assigns a TAEN score combining the frequency and recency of collaborations. Finally, based on the severity of the report, a fixer team comprised of existing and new developers is allocated using the TAEN score and current workload of developers.

For assessing the compatibility of TAEN, it is applied on Eclipse JDT, AspectJ and Netbeans. The experimental analysis shows that TAEN successfully allocates 78.55%, 67.18% and 51.27% actual developers in the suggested teams for Eclipse JDT, Netbeans and AspectJ respectively which are higher than an existing technique known as KSAP. Besides, the number of workloads assigned to the new and existing developers is calculated, which depicts that TAEN is successful in assigning tasks to new developers, whereas KSAP fails to do so.

Acknowledgments

"All praises are due to Allah"

First of all, I praise Allah, The Almighty and The Lord of The World, for giving me this opportunity and granting me the ability to continue my research work effectively.

I take this opportunity to express my significant appreciation and profound respect to my thesis supervisor, Professor Dr. Kazi Muheymin-Us-Sakib, Institute of Information Technology, University of Dhaka. Without his patience, support, motivation and immense knowledge, this research could not be successful. His continuous guidance while doing the research and writing the thesis has enabled me to complete it successfully.

I would like to thank my thesis reviewers Dr. Muhammad Masroor Ali and Dr. M. Lutfar Rahman for their valuable feedbacks which has enabled to enhance the quality of the thesis.

I would like to pass on my ardent appreciation to all faculty members, Institute of Information Technology, University of Dhaka, for their support, motivation, criticism and productive feedback which has enormously reinforced my confidence during my thesis.

I am also thankful to Ministry of Posts, Telecommunications and Information Technology, Government of the Peoples Republic of Bangladesh for granting me ICT Fellowship No - 56.00.0000.028.33.065.16 (Part-1) -772 Date 21-06-2016.

List of Publications

1. "A Team Allocation Technique Ensuring Bug Assignment to Existing and New Developers Using Their Recency and Expertise" in *Proceedings of the 3rd International Conference on Advances and Trends in Software Engineering (SOFTENG)*, pp. 96-102, Venice, Italy, April 23-27, 2017.

2. "A Bug Assignment Technique Based on Bug Fixing Expertise and Source Commit Recency of Developers" in *Proceedings of the 19th IEEE International Conference on Computer and Information Technology (ICCIT)*, pp. 592-597, Dhaka, Bangladesh, December 18-20, 2016.

3. "A Bug Assignment Approach Combining Expertise and Recency of Both Bug Fixing and Source Commits of Developers" in *17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2017. *(Submitted)*

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the increasing number of reported bugs, bug assignment has become a crucial task for software quality assurance. Studies show that in open source projects about 50 to 60 bugs are reported on a daily basis [6]. The most important phase of bug assignment is identifying appropriate developers who can best resolve the bug. However, huge number of developers generally work parallel in distributed setup for developing the software. Selecting developers from such a large group is time-consuming and error-prone. Besides, software development is a team work, which turns bug resolution into a collaborative task. It is reported that Eclipse bug reports involve on average teams of ten developers [4]. Few existing techniques suggest expert team of developers using past bug reports. However, the past reports do not contain any trace of the newly joined developers. So, these techniques fail to include the new developers in the final suggestion. This leads the existing techniques to improper task allocation among developers. As a result, the new developers fail to gain and share knowledge through the bug resolution process. In this essence, an automatic technique can leverage bug assignment by identifying a fixer team considering the new developers. Keeping this objective in mind, this thesis proposes an automatic team allocation technique by ensuring bug assignment to both existing and new developers.

This chapter first demonstrates the issues of automatic team allocation and describes the motivation of this research work. It then formulates the research question and proposes a guideline towards answering this question. It also briefly describes the contribution and achievement of this research. Finally, the organization of this thesis is outlined for giving a guideline to the readers.

1.1 Motivation

A truth about software testing denotes that there is no way to prove that a program or system of any reasonable size is "bug free" [7]. As bugs are part of software life, resolution of bugs is imperative for assuring software quality. Whenever bugs are reported into the bug tracking system, it is the task of a bug triager to analyse the bug, and assign it to potential developers for faster fixation. However, manually assigning bug reports incur considerable amount of time and cost. For example, more than 333,000 bugs were reported for Eclipse project from October 2001 to December 2010, in average 99 bugs per day [7]. If it takes minimum 5 minutes to triage a bug, over 8 hours would be spent on bug assignment alone. According to Tyler Downer[1], a former Mozilla Community Lead, there were 5934 unconfirmed bugs in a shipping version of Firefox 4. Among these reports about 2598 had not been touched over 150 days since the launch of Firefox 4. These examples raise the need of automatic bug assignment.

Bug assignment tends to suggest a list of fixers while arrival of a new bug into the system. However, delegating bug reports to developers is a difficult task. The reasons are outlined as follows. Firstly, a huge number of developers work concurrently for developing the software. Manually identifying actual fixers from these developer's group is a tedious and error-prone job. Secondly, a developer may shift to other company or project, and new developers may join at any

[1]http://www.somethingawful.com/news/bugs-of-firefox/

time. Thus, combination of both previous and current activities needs to be considered for effective triaging. Again, a developer may not be interested in a randomly assigned bug, which may lead to unnecessary reassignments and prolonged fixing time. These difficulties while bug triaging, raise the need for automatic bug assignment techniques.

Automated bug assignment techniques assign tasks based on the bug fixing profile of the developers. These profiles are generated from either source code commits or bug fixing history. When a new bug report arrives, its keywords are matched with the source or bug report information. The developers who are associated with the most similar source or bug reports, are suggested as list. However, consideration of these individual information sources may result in inexperienced and inactive developer suggestions. The major limitation of these approaches is the ignorance of collaboration information among developers. As a result, these approaches fail to suggest team fixers.

In order to suggest developer teams, the collaborations among the developers need to be considered. Few team-based assignment techniques are also proposed in the literature. These approaches determine developer expertise and communication profiles using only previous fixed bug reports. The past bug reports do not contain any trace of recently joined developers. Therefore, these approaches fail to delegate tasks to new developers. With the passage of time, new developers may join the company. These developers may also come with different levels of skills. Associating new developers in bug fixing process may enable the new developers to participate in fixing process and gain knowledge.

New developers can be included in the team allocation process by extracting their expertise in the form of initial bug solving preference. This would mitigate the chances of reassignments, resulting from random assignments. It may also enable incremental profile building of new developers by solving bugs. However, none of the existing techniques allocate fixer teams considering new developers.

Hence, the need of an automatic expert and recent team allocation technique utilizing both existing and new developers is still a research issue.

1.2 Issues in State-of-the-Art Approaches

In the literature, a rich collection of individual bug assignment approaches have been devised. For reducing the cost of manual bug triage, Cubrani et al. first proposed the problem of automatic bug triage in a form of text categorization based technique [8]. Text categorization based techniques build a model that trains from past bug reports to predict the correct rank of developers [6, 9–12]. Baysal et al. have enhanced these techniques by adding user preference in the recommendation process [10]. The framework performs its task using three components. The *Expertise Recommendation* component creates a ranked developer list using previous expertise profiles. The *Preference Elicitation* component collects and stores a rating score regarding the preference level of fixing certain bugs through a feedback process. Lastly, knowing the preference and expertise of each developer, the *Task Allocation* component assigns bug reports. The applicability of this technique depends on user ratings, which can be inconsistent. Besides during recommendation it does not take new developers into account. As a result, improper workload allocation among developers may occur.

CosTriage, a cost aware developer ranking algorithm, has been developed by Park et. al [13]. The technique converts bug triaging into an optimization problem of accuracy and cost, which adopts Content Boosted Collaborative Filtering (CBCF) for ranking developers. As the input to the system is only previous bug history, the technique contains no trace regarding new developers to assign tasks.

Source based bug assignment techniques have also been proposed. Matter et. al have suggested DEVELECT, a vocabulary based expertise model for recommending developers [12]. The model parses the source code and version history

to index a bag of words representing the vocabulary of source code contributors. For new bug reports, the model checks the report keywords against developer vocabularies using lexical similarities. The highest scored developers are taken as fixers. Another source based technique has been proposed in [1]. The technique first parses all the source code entities (such as name of class, attributes, methods and method parameters) and connects these entities with contributors to construct a corpus. In case of new bug reports, the keywords are searched in the index and given a weight based on frequent usage and time metadata. The main limitation of these techniques is, it suggests novice developers without considering their experience and preference. As these techniques require minimum one source commits, these also fail to include new developers in final suggestion.

Vaclav et al. have presented a study to compare the trend of bug assignment in the open source and industrial fields [14]. The study applies Chi-Square and t-test for evaluating the variability of those two fields dataset, and reports identical trends in terms of distribution. Most importantly, it concludes with some findings highlighting the need for task assignment to individuals and team recommendation.

There are few team assignment techniques, which are also proposed in the literature. Zhang et al. developed a team assignment technique called KSAP [4]. It initially constructs a heterogeneous network using existing bug reports. When a new bug report arrives, the technique applies cosine similarity between the document vectors of new and existing reports to extract the K nearest similar reports. Next, the commenters of these K similar bugs are taken as the candidate list. Finally, the technique computes a proximity score for each developer based on their collaboration on the network. The top scored Q number of developers are recommended as fixer team. Although this technique can meet the need of team recommendation, it fails to cover the requirement of task allocation to new developers. These related works show that, none of the existing approaches consider

new developers while allocating teams to resolve above mentioned limitations.

1.3 Research Question

Considering the limitations of the above mentioned approaches, the objective of the research is defined. The objective is to propose an expert and recent team allocation technique by ensuring task assignment to both existing and new developers. To suggest an expert and recent developers team, first the procedure of combining the recency and expertise information needs to be determined. Therefore, firstly an expertise and recency based bug assignment approach has been devised. Next, this concept of expertise and recency combination is applied to propose a team allocation technique by considering the new developers. Keeping these factors in mind, the following research questions have been formulated -

1. How to accurately assign newly arrived bugs to potential developers, based on developers' recent activity and bug fixing experience?

   Existing bug assignment techniques consider either recent activities or previous fixes while identifying potential fixers. As a result, the accuracy of these techniques may degrade due to suggesting inexperienced and inactive developers. So, the expertise and recency information are combined for allocating developers, which leads to two more sub-questions -

   (a) How to extract developers' current and previous activities using commit logs and bug tracking system respectively?

   Source code entities represent developer vocabulary. So, the current activities can be collected from the source code commits performed by the developers. Again, the higher number of times developers work on specific bugs, the higher experience they gain. Thus, the *summary* and *description* fields of the bug report can be used to

extract expertise information. Both of these information sources need to be pre-processed and indexed for repressing expertise and recency information.

(b) How to connect developer activities with new bug report to accurately suggest potential developer list?

In order to find potential fixers, the similarity between the bug report and developer activities needs to be determined. To do so, the keywords of the new report are needed to be searched on the constructed index and tf-idf term weighting technique can be applied on the search results to measure an expertise and recency score for each developer. Finally, the highest scored developers can be suggested as appropriate fixers. The accuracy of the suggested lists need to be evaluated using metrics such as - Top N Rank, Effectiveness, Mean Reciprocal Rank (MRR) etc.

2. How to allocate an experienced and recent developers' team by ensuring task assignment to new developers?

(a) How to determine the initial bug solving preference of new developers?

For identifying the bug solving preference of new developers, first, the possible types of available bugs are needed to be determined. Latent Dirichlet Allocation (LDA) topic modeling can be applied to determine possible types of bugs. The developers who worked on similar types need to be grouped together. Finally, the representative keywords and bug reports can be presented to the new developers to collect their initial bug solving preference, and add them in the developer groups.

(b) How to extract previous and recent collaboration information among developers?

To extract developers collaboration skill while solving bugs, a collaboration network is needed to be constructed using fixed bug reports. On arrival of new bug reports, its bug type needs to be identified. The developer group who have worked on this identified bug type, needs to be determined. Developers generally collaborate on same bugs or bugs of same components. So, how frequently and recently the identified developer collaborate can be extracted by parsing the network. Based on these extracted collaboration, a score can be calculated for each of the developers representing their recent and expert team collaboration skills.

(c) How to ensure bug assignment to both new and existing developers?

All the reported bugs in the bug tracking system are not equally severe. If the reported bug is a severe one, top scored developers are allocated as the bug may require complex fixing. Otherwise, the bug is assumed to be a normal one and a team composed of both new and top scorer developers are allocated. The validity of the suggested teams can be evaluated by calculating the recall of the teams. Moreover, the bug assignment to new developers can be determined by measuring the number of workload or bug reports assigned to the developers.

1.4 Contribution and Achievement

To answer the above research questions, this research proposes an expert and recent team allocation technique by ensuring task assignment to existing and new developers. The contributions are summarized below.

Firstly, this research combines the previous bug fixing expertise and recent source commits of developers to accurately assign bug reports. This combination is performed using three steps. It first takes source code and commit logs as

inputs. To represent the time of source entity usage, it builds an index that maps the source code identifiers with developer commits. Next, the technique uses the fixed bug reports to construct another index connecting bug report features (keywords) with the report fixer. Finally, when new bug reports arrive, the final step extracts the new report keywords, and queries these keywords on the above mentioned built indexes. By applying tf-idf term weighting technique on the query results, a score called ERBA is assigned to each developer. The high scored developers are recommended as appropriate fixers. This overall approach answers the first research question mentioned in Section 1.3.

An experimental analysis is performed on three open source projects for evaluating the compatibility of the above mentioned approach. The open source projects are Eclipse JDT, AspectJ and SWT. The results show that the proposed technique accurately retrieve about 90% of the actual fixers whereas an existing technique [1] retrieves only about 65% of the fixers. The results also depict that the proposed technique shows the first relevant developer at a higher position (for Eclipse at 2.05) than the existing one (8.1). Besides, for each of the projects an improvement on the Mean Reciprocal Rank value is also achieved (for example - for Eclipse 13%).

In the next phase, an expert and recent team allocation technique has been proposed to ensure task assignment to both existing and new developers. The technique first takes fixed bug reports as input and applies the Latent Dirichlet Allocation (LDA) technique on these reports to determine the possible bug types. The developers who worked on similar bugs are then grouped under the bug type. These bug reports are also used for generating a developer collaboration network in this phase. The network contains four types of nodes and eight types of edges which are extracted from the reports. The current workload of each developer is also determined using the input reports. Next, on arrival of new developers, it elicits their bug solving preference by presenting them with main

representative terms and reports of each bug type. The new developers are then initially grouped under the types they choose. For incoming reports, it extracts the *summary*, *description* and *severity* properties and determines their bug types to identify the initial fixer group. For suggesting a team, the frequent and recent collaborations among the developers of the initial fixer group are extracted. Based on these collaborations, a score is assigned to each developer. While score calculation, the earlier proposed concept of recency and expertise combination is considered. Finally, based on the *severity* of the incoming reports a team of developers is suggested using the score and current workload of developers. This technique answers the second research question.

For evaluating the compatibility of the team allocation technique, it is applied on three open source projects - Eclipse, Netbeans and AspectJ. The results are compared with an existing team allocation technique [4]. The results show that the proposed technique achieves 78.55%, 67.14% and 61.27% recall on retrieving the actual fixers in the allocated team for Eclipse, Netbeans and AspectJ respectively, which is higher than the existing technique. The workload assigned to each developer by the team allocation technique are also measured and presented. The results also show that the proposed technique successfully assigns tasks to the new developers, whereas the existing technique totally ignores them.

1.5 Organization of the Thesis

This section gives an overview of the remaining chapters of this thesis. The chapters are organized as follows -

- **Chapter 2: Background Study**

  In this chapter, the preliminary topics for understanding the proposed technique have been described. The basic ideas on software bug, bug reports, bug tracking system are also introduced. Along with this, a general archi-

tecture of automated bug assignment has also been presented.

- **Chapter 3: Literature Review**

  This chapter demonstrates the existing bug assignment techniques. The pros and cons of each of the existing techniques are also described for better understanding the open research issues.

- **Chapter 4: Expertise and Recency Based Bug Assignment**

  This chapter introduces a technique for combining the source commit recency and bug fixing expertise of developers in bug assignment. The experimental results for supporting the technique have also been presented here.

- **Chapter 5: Team Allocation Considering New Developers**

  A team allocation technique for ensuring bug assignment to both existing and new developers (using the concept discussed in Chapter 4) has been devised in this chapter. The details of the experimental set up and result analysis for assessing the proposed technique is also presented here.

- **Chapter 6: Conclusion**

  This chapter first summarizes the overall thesis. It then discusses the threats to validity of the proposed technique. Lastly, the chapter concludes the thesis with considerable future remarks.

Chapter 2

Background Study

Software bugs are inevitable. A software bug is generally referred as a software failure, which may result due to errors in source code, design, components or the supporting environment. A bug interrupts the normal execution of the software system. So, resolving bugs is an essential step of software maintenance and development. Software maintenance studies indicate that maintenance costs are at least 50%, and sometimes more than 90%, of the total costs associated with a software product [15]. Whenever a bug occurs, it is generally reported to the development team for faster resolution. These reported bugs are accumulated and maintained by various bug tracking systems, such as Bugzilla [16]. After reporting of a bug, the bug reports are analyzed and assigned to developers for fixation. The developer who is responsible for analyzing the bug report is known as bug triager. The bug triager then manually analyzes the bug report and finds an appropriate developer for resolving the bug. However, manually identifying potential fixer is error-prone and a labour intensive work. Any mistake in assignment can lead to reassignment and prolonged bug fixing time. In this essence, automatic bug assignment can assist the bug triager by suggesting a fixers list without any human intervention. Therefore, in this chapter the elements associated with automatic bug assignment are introduced for better understanding

bug triaging. The widely used techniques to leverage automated bug assignment are also described.

2.1 Overview of Software Bug

Dealing with software defects or bug is a major part of software product maintenance [17]. With the increasing size and features of a software system, the number of reported bugs also increase. For example - about 200 bugs were reported daily for Eclipse near the release dates [9]. In order to maintain the quality of the software, bug resolution is essential. The first step of bug resolution is assigning the bug report to an appropriate developer. This assignment is performed by analyzing information from software source code, commit logs and bug reports. In the following sub sections a brief view of software bugs is discussed, followed by the severity of the bugs. Along with this, the details of bug reports, bug repository system, bug reporting procedure are also explained.

2.1.1 Software Bug

A bug is simply a behaviour deviation between the specification and the implementation. It causes a computer program or system to produce unexpected results or behavior [18]. In industrial software development, a number of developers work parallel to meet the product release deadlines. The hurry to meet the deadlines generally result in ignorance of standard development scheme which may lead to software bugs. Besides, in open source systems, developer works in distributed setup to contribute in the system. As a result, ideal coding practices may be ignored. Ignorance of standard practices creates errors in the software code and any tiny coding error is referred to as software bug.

Software bugs generally degrade the quality of the system. When a system fails to meet the needs of users, the system is said to contain buggy entities.

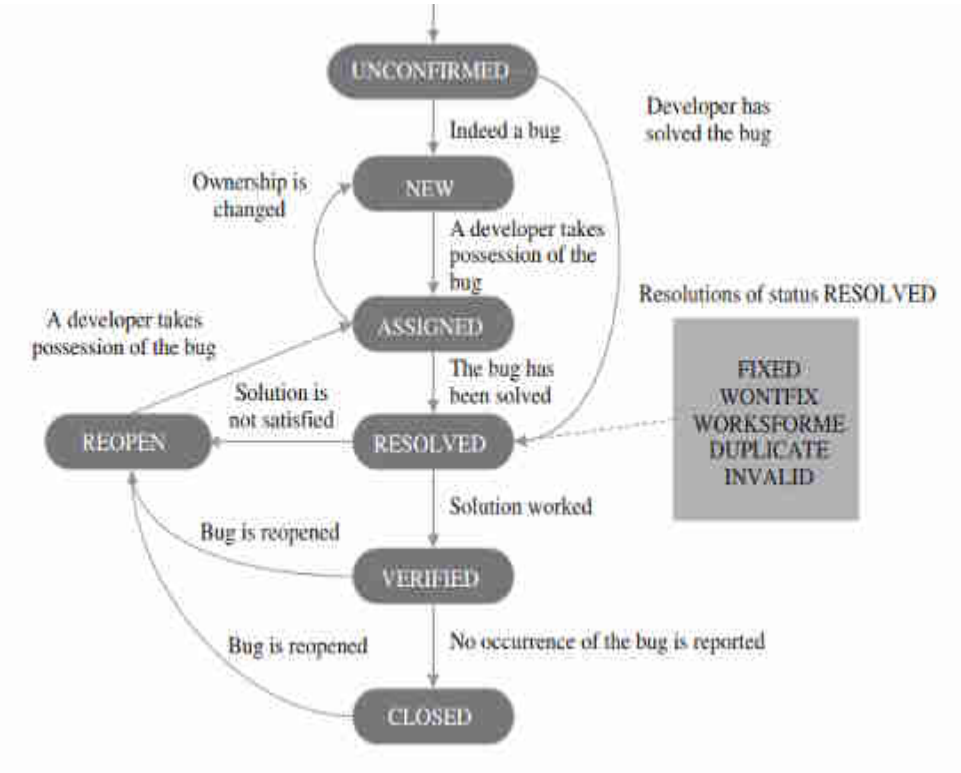

Figure 2.1: Life Cycle of a Bug Report [5]

If the software exhibits unexpected behaviour, it incurs cost both in terms of maintenance and customer satisfaction. A survey by the National Institute of Standards and Technology estimated the annual cost of software bugs which is around $59.5 billion [19]. So, finding and fixing bugs has become an essential step for maintaining quality software.

2.1.2 Life Cycle of Software Bug

Bug life cycle is referred to as the journey through which the bug goes during its lifetime. A bug goes through several bug *status* from reporting to the final resolution of the report. Figure 2.1 shows the steps through which a bug report passes during its life cycle. Each of these steps are described below -

- **Unconfirmed:** When a bug is created, initially it gets the default status as *Unconfirmed*. It is so because, the bug has not been analyzed or checked by any bug triager.

- **New:** While assigning bugs, the first step a bug triager performs is to analyze the bug. This analysis is performed because developers can concurrently report bug issues. A same issue can be reported by multiple developers. Again, the reported problem can be a performance issue rather than real bugs. So, the triager first analyzes the bug to identify if it is duplicate or is indeed a new bug. If the bug is identified as a real issue, then the triager sets the *status* of the bug to *New*. Moreover, if the bug is a duplicate of any solved bug, then its *status* is set to *Resolved* as shown in Figure 2.1.

- **Assigned:** The triager then finds an appropriate developer or developers to allocate the bug for fixing. Finding an appropriate developer for a new bug is a crucial task, as wrong assignment induces cost. Once the report is allocated to potential developers, its status is changed to *Assigned* by the triager.

- **Resolved:** Once a developer gets possession over a reported bug, the developer performs several tasks for solving the bug. For example - the developer may reproduce the bug, localize it and perform necessary code change to fix it. After that, the developer may change the *status* of the bug as *Resolved*. During this stage, the bug may have different bug resolution *status* as shown in the right part of Figure 2.1. If the necessary code changes fix the bug correctly, then the resolution is set to *Fixed*. If the reported bug can not be solved, it is marked as *WontFix*. The bug is tagged as *WorksForMe* when the final resolution is possible provided that, additional details of the bug is required. If the bug is found similar with previously reported bugs, then it is marked as *Duplicate*. Lastly, if the bug is identified as an invalid issue, its resolution *status* is changed to *Invalid*.

- **Reopen:** The resolved bugs are next sent to the testers for checking its

resolution. If the bug exists even after the resolution or the testers are not satisfied with the resolution, the bug is reopened for better fixation. Thus, the bug is tagged as *Reopened* and assigned to same or different developer for resolution.

- **Verified:** If the testing team finds the bug resolution is correct, then the bug is marked as *Verified* bug. A verified bug can also be reopened, if any instance of the bug is found in future.

- **Closed:** A verified bug is set to *Closed* bug, until any occurrence of the bug is found.

2.2 Bug Report

A bug report is a software document which describes the features of software bugs and is submitted by a developer, a tester, or an end-user [20]. It is also known as "Defect Report" [21], "Fault Report", "Failure report","Error Report", "Problem Report" etc. As stated before, a bug report may be submitted by testers and general users as well. However, the general users and QA engineers do not have detailed idea about source implementation. If the features of a bug is not reported properly, it may become difficult for the developers to identify the bug. The structured form of bug reports help to meet the gap between the bug reporters and developers.

2.2.1 Bug Report Attribute

Bug reports are generally submitted by filling structured forms. Figure 2.2 shows a sample bug report form used by bug tracking system, Bugzilla [16]. While reporting the bug report, the submitter needs to fill up a number of attributes. These attributes help the developers to understand the characteristics of the

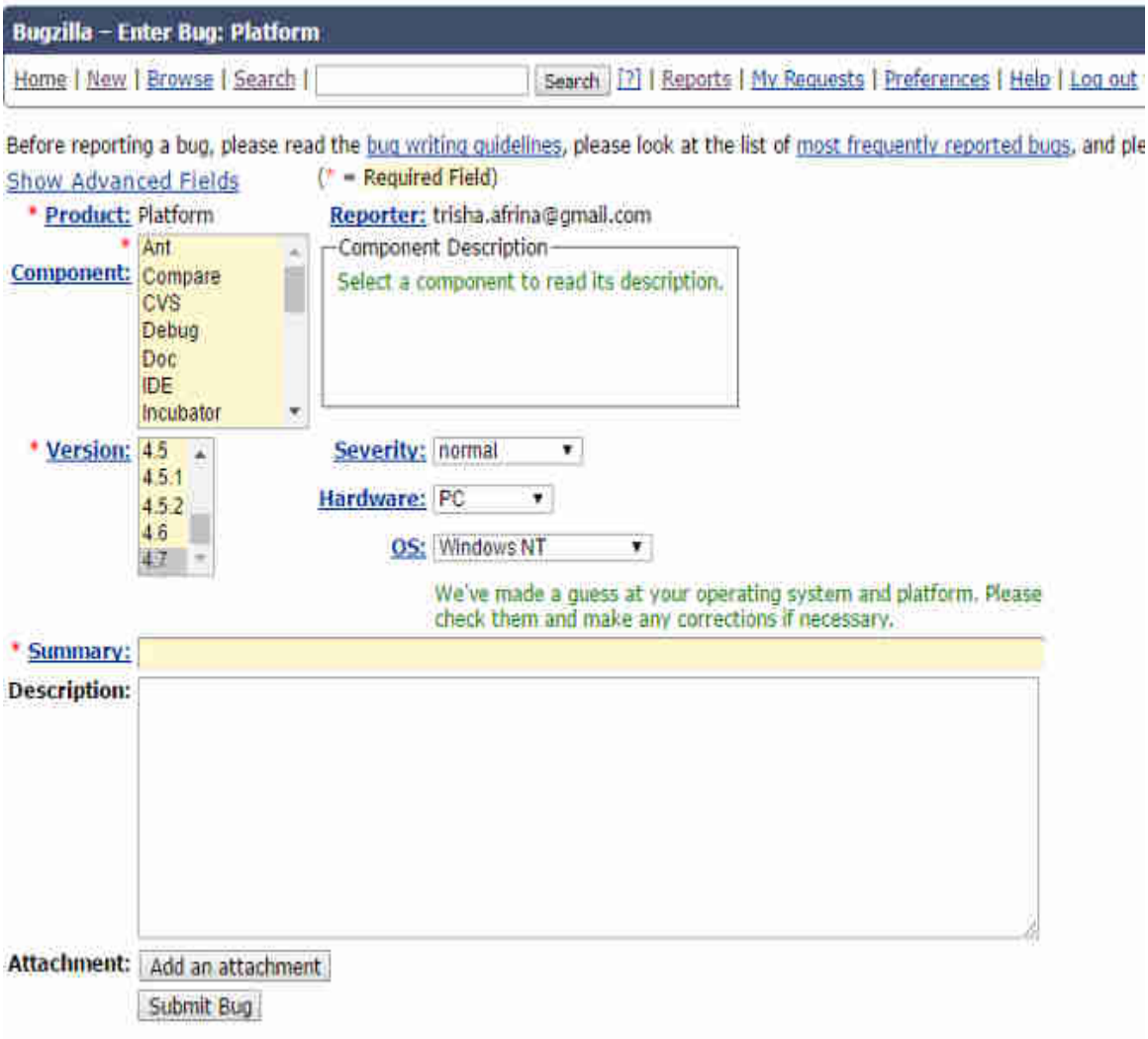

Figure 2.2: Sample Bug Reporting Form of Bugzilla

report. Few attributes of a bug report are as follows -

- **Summary:** This field is a short sentence which succinctly describes what the bug is about. It generally shows the title of the bug report.
- **Description:** It describes the problem in detail. The reporter may specify actual bug location, failed functionalities, execution traces etc in this field.
- **Product:** The product to which the bug report corresponds is specified here as a number of products are developed under same organization.
- **Component:** A number of components are implemented to support a

Figure 2.3: Sample Bug Report of Eclipse Bug#92009

software product. This field specify the component of the product which contains the bug.

- **Source Attachment:** Test cases, patches or source portions can be attached here for specifying the actual bug location in details.

- **Severity:** This field indicates how severe the problem is. For example - the problem can be associated with serious application crash, minor cosmetic issue or enhancement requests. Total seven types of severity value are available to choose by the reporter. The details and specification of these attribute values are enlisted in Table 5.1 of Chapter 5.

- **Version:** The version field is usually used to indicate the products version in which the problem occurred.

Usually, these attributes are filled by the reporters while initially submitting the bug report. However, a number of entities are also added to the report throughout its resolution process. Some of these entities related to this research work are outlined in following section.

Figure 2.4: Comment Section of Eclipse Bug #92009

2.2.2 Bug Report Contributor

From the submission of a bug report towards its resolution, a number of developers contribute to the report. Based on their activities, the developers can be referred to as reporter, commenter, fixers etc. Figure 2.3 and 2.4 shows partial portions of a *fixed* bug report of Eclipse #92009. The right upper part of Figure 2.3 indicates that the bug is reported by "Dejan Golzic". Figure 2.4 shows that 6 comments are performed by three developers. Here, developers "Wassim Melhem" and "Dejan Golzic" commented to inform about a bug regarding a *Finish Button*. However, the comments of "Dani Megret" indicates that the developer

Who	When	What	Removed	Added
daniel_megert	2005-04-20 12:28:38 EDT	Assignee	jdt-text-inbox	daniel_megert
		Priority	P3	P1
daniel_megert	2005-04-20 12:39:49 EDT	Status	NEW	RESOLVED
		Resolution	---	FIXED
		Summary	Externalize Strings wizard broken	[nls tooling] Externalize Strings wizard broken
daniel_megert	2005-04-22 12:34:56 EDT	CC		dsp
daniel_megert	2006-06-07 09:56:47 EDT	Target Milestone	---	3.1

Figure 2.5: History Information of Eclipse Bug #92009

is working toward resolving the bug as well as guiding the other developers on how to use the fix.

It is showed in Figure 2.1 that the *status* of the bug report changes, when developers work on the bug. Whenever developers commit changes on the bug report, these changes are recorded by the bug maintenance systems. The full resolution history of a bug report is also stored in the report. Figure 2.5 shows the full fixing history of Bug #92009. The records of the table shows that "Dani Megret" take possession over the bug from developer "jdt-text-inbox". He then sub-subsequently set the priority of the report, resolves it and changes the *status* of the bug report from *Resolved* to *Fixed*. By analying this activity history, the actual fixer set of the bug report can easily be identified. For example - developers "Wassim Melhem" and "Dejan Golzic" commented on the bug, but did not work for resolving it. Thus, they are the only collaborators of the report. That is, all commenters can not be fixers. The actual fixers can be identified by analyzing bug resolution history. Here, "Dani Megret" is both commenter and fixer of the report.

2.3 Bug Tracking System

A Bug Tracking System (BTS) manages and stores the bug reports submitted by developers. These systems provide users with structured and generalized forms to submit bugs (see figure 2.2). To a large extent, bug tracking systems serve as the medium through which developers and testers communicate for resolving the bugs [22]. It not only manages the bug reports, but also stores developer coordination and expertise information [8]. In both industry and open source projects, the bugs are tracked using different BTS such as - Bugzilla, Jira, Mantis, Trac etc. These systems are particularly important in open source software development, where the team members can be dispersed around the world. In such widely-distributed projects, the developers and other project contributors may rarely see each other. In these case, BTS play vital role in developer collaboration for resolving bug reports.

2.4 Bug Assignment

Bug assignment refers to the process of automatically assigning the reported bugs to appropriate developers [23]. Studies have shown that, depending on the phase of the development cycle, developers spend 50% [24] to 70% [25] of their time for communicating with or identifying actual fixers. In software projects, huge number of developers work parallel with different groups. Manually identifying appropriate developers from these groups is time-consuming and error-prone. It is reported that during June 2004 to June 2005, 13016 reports were filed, averaging 37 reports per day, with a maximum of 220 reports in a single day [26] .If any triager spends minimum 5 minutes to read and handle each report, three person-hours per day is required on average triaging all the bug reports. Moreover, mistake in bug assignment generally leads to unnecessary reassignments and prolonged fixation. Besides, with the passage of time, developers may leave

the company or new developers may join the group. Ignorance of these situations may affect the effectiveness of bug assignment techniques. Thus, making bug assignment a more challenging task.

Automatic bug triaging can mitigate the problems of manual assignment. It not only reduces time and labour cost, but also assigns the bug reports to accurate developers. Various bug assignment techniques have been proposed in the literature. For assigning bug reports the techniques first collect and preprocess the available information sources - such as source code, commit logs and bug reports. These information sources are then stored in program readable formats for use when new reports arrive into the system. On arrival of new bug reports, the information of the new bug report are searched in the previously stored data. Finally a list if developers are suggested using various Information Retrieval (IR) based techniques. The general procedure of bug assignment can be divided into few steps. The steps are-

- Data Collection
- Indexing and Graph Construction
- New Bug Report Processing
- Ranking

In the following sections, each of these steps are described briefly.

2.4.1 Data Collection

When developers work on a software product, they accumulate knowledge and expertise. In order to assign bugs, the expertise history of developers on solving various bugs needs to be determined. While assigning bug reports, this information are collected from system source code, commit logs and bug reports. During

the lifetime of a software project a large amount of historical information is collected and stored by version control systems such as CVS [27]. Therefore, the information stored in the commit log play important role while finding potential fixers. This information help to identify developer who has the most expertise in the techniques and skills required for a task, and thus, would produce the best result in the best possible time [28]. Merging the source code information along with the commits enables extraction of developer vocabularies [12].

Version information can be enhanced with data from bug tracking systems that report about past maintenance activities. Bug reports typically contain a detailed description of the problem in natural language text, which helps in automatic task assignment [29]. Anvik et al. performed an empirical study to determine implementation expertise from the data in source and bug repositories [30]. The study found that both source and bug history had low to moderate precision, ranging from 39% to 59%, and high recall, ranging from 71% to 92% while identifying actual developers. Therefore, almost all bug assignment techniques start processing by collecting information from source commit history and bug repositories.

The commit logs and bug reports contain narrative textual formats. Again the source entities are composed by combining keywords. So, all these information sources need to be pre-processed to avoid redundant and noisy terms. Few most commonly used pre-processing steps in automatic bug assignment are described below.

2.4.1.1 Stop Word Removal

Modern information retrieval techniques use a range of statistical and linguistic tools for effectively analyzing the textual content of documents. One such standard technique used in text categorization is stop word removal [31]. Stop word refers to the set of non-informative words in a document, which play no

role in determining the type or topic of the document. Few example of stop words are - articles, prepositions, conjunctions etc. Existence of stop words in a document reduces the recall of text categorization techniques. Besides, the stop words have no potential to distinguish between categories. So, the stop words should be filtered while pre-processing the documents [32].

In the field of bug assignment, the content of a bug report is generally analysed for understanding the characteristics of the report. It is stated before that the *summary* and *description* property of bug reports are submitted in textual forms. This textual description may contain a number of stop words. So, for determining the characteristics of bug report, the stop words contained in the *summary* and *description* field of the report are removed. Stop words can be language and task dependent. However, there is a set of general keywords which are considered as stop words in all scenarios. Some example of such stop words are as follows - "against", "all", "am", "an", "and", "any", "are", "aren't", "as", "at", "be", "because", "been", "before", "being" etc.

2.4.1.2 Stemming

One of the main problems in text categorization is the fact that a single word may occur in some (or many) different forms [33]. In order to mitigate this problem, another text pre-processing technique known as stemming is commonly used. Stemming refers to the process of reducing inflected words to their stem, base or root form. The basic idea is to strip off affixes and leaves with a stem in order to achieve the correct goal most of the time [34]. For example - the three words, "connects","connected" and "connection" are all mapped to the same stem "connect". Stemming helps to enhance the recall of text search by reducing inflection information of words.

Various stemming algorithms are developed by researchers to covert inflectional form of keywords towards their root form. Lovins Stemmer, Paice Stemmer

and Porter Stemmer [35] are such popular stemmers. The best known stemmer of linguistic sort is Porter [36], which was initially developed for English. Porter's approach was later extended to French, Italian and Dutch. It has repeatedly been shown to be empirically very effective while analysing text documents [37]. Porter's algorithm contains five phases of word reduction which are applied sequentially. Each phase is divided into several different rules, and the rule with longest matching suffix is then applied to the input words. For example, the first phase consists of following four rules :

Rule: SSES → SS , **Example:** caresses → caress

Rule: IES → I , **Example:** ponies → poni

Rule: SS → SS , **Example:** caress → caress

Rule: S → , **Example:** cats → cat

Here, the first rule specifies that if a word owns a suffix similar to "sses", it will be replaced by "ss". For example, the word "caresses" will be converted into caress. Therefore, the keywords of *summary* and *description* fields of the bug report are stemmed using Porter's english stemmer to increase the effectiveness of reports type identification.

2.4.1.3 Keyword Decomposition

Object Oriented Programming (OOP) language follows some established conventions to name the source code entities or keywords such as - variables, methods, class, packages, interface etc. The most used naming practice used in OOP language is Camel Case. The Camel Case naming convention refers compound words, where words are written without using spaces and each word begins with a capital letter. It is also known as Upper Camel Case. A variation of Camel Case is Lower Camel Case where, the first letter of the first word is lowercase. Also in some cases, keywords are detached using separator characters such as - dot, underscore, dash, percentage, dollar etc. As bug reports generally indicate

Table 2.1: Example of Keyword Decomposition Procedure

Decomposition Based On	Keywords	Processed Keywords
CamelCase Letter	getUserList	get, user , list, getuserlist
Seperator Character	get_Users	get, users
	get-Users	get, users
Number and Special Character Removal	getUser1	get, user, getUser

the buggy potions of source code. Hence, the bug reports may contain above mentioned composed keywords. For better understanding the topic of the bug reports and commits, such keywords needs to be decomposed.

The keyword decomposition is done based on CamelCase letter, separator character, number and special character removal. Some example case of keyword decomposition used while bug report processing is shown in Table 2.1.

2.4.1.4 Language Specific Keyword Removal

Each programming language has some predefined keywords. Few such keywords of java programming language contain - "public", "private", "int", "String", "switch", "if", "for", "while" etc [38]. The bug reports may contain those language specific source code entities or source portions. These keywords play no role while identifying the expertise of developers. Instead, these keywords add irrelevant information while determining the type of the reports. So, the keywords related to the language of the input source code, are removed form the *summary* and *description* field of the bug reports.

2.4.2 Indexing and Graph Construction

Once all the required data are collected and pre-processed, these information needs to be stored in a structural formats for faster retrieval. Various types of data processing techniques have been used in literature. For example - few studies index the information of bug reports for searching similar keywords when

Table 2.2: Sample Documents

DocId	Terms
Doc1	crash, ui, method
Doc2	ui, file, view
Doc3	ui, file, method, button

new reports arrives ([1, 13]). On the contrary, [6, 39, 40] constructs graph using the information of bug reports for extracting developer collaborations. In the following some of these data handling techniques are discussed.

2.4.2.1 Lucene Indexing

Lucene is a full-text search library which adds search functionality to an application or website [41]. It provides faster search responses. The reason behind this higher performance is that instead of searching the text directly, Lucene searches an index. This is similar to retrieving pages in a book related to a keyword by searching the index at the back of a book, rather than searching the words in each page of the book. This type of indexing is called an inverted indexing, because it inverts a page-centric indexing to a keyword-centric indexing. The main steps in inverted index construction is as follows -

1. Collecting the document to be indexed
2. Tokenizing the text (which is done using various processing step mentioned in 2.4.1
3. Performing linguistic preprocessing of tokens
4. Indexing the documents which maps each keyword to a list of documents where the keyword has appeared

The core functionality of Lucene is to index the informations. A overview of Lucene indexing process is given in Figure 2.6. The basic unit of Lucene indexing is Documents. Documents are collection of fields, where each field has an

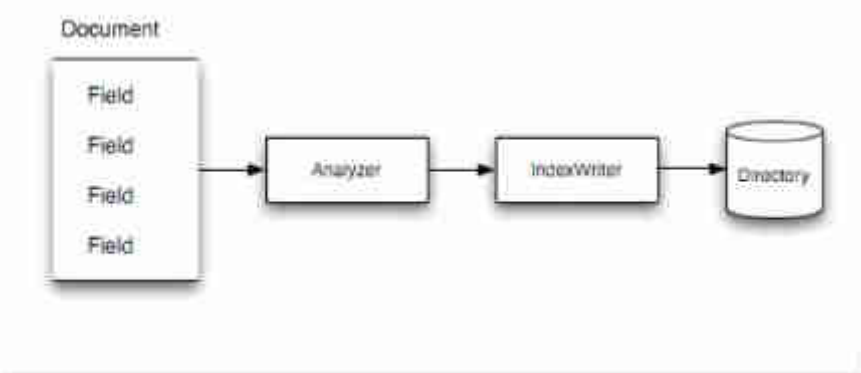

Figure 2.6: Sample of Lucene Indexing Process

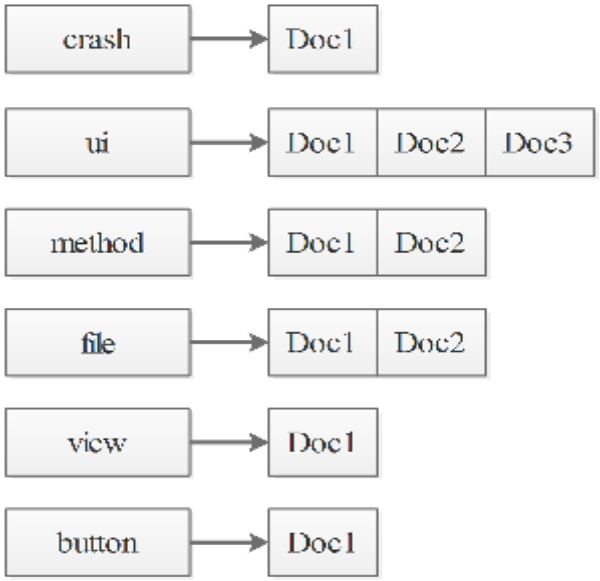

Figure 2.7: Inverted Index for Table 2.2

unique name and field value. A number of field properties are also set before indexing to specify the possible searching criteria. These documents are then analysed by Analyzers before creating the indexes. The IndexWriter then performs the ultimate task of creating, opening or editing indexes as required. The indexes can be stored or updated into file system or RAM Directory based on the setting of the IndexWriter. Various bug assignment techniques creates indexes of documents, where the source or report keywords and developer names are set as document fields to search developer expertise or vocabulary [1].

Let *Doc1*, *Doc2* and *Doc3* are the three documents available in the document collection. These documents along with their contained terms are given in Table

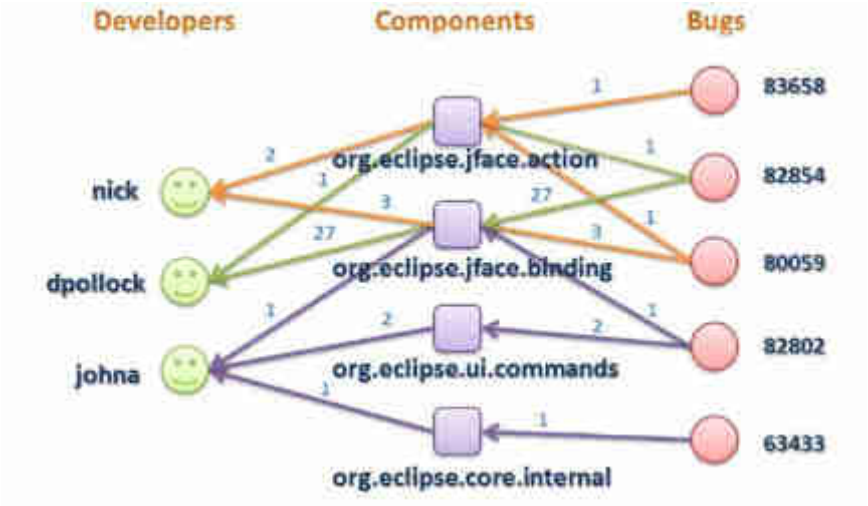

Figure 2.8: Sample Developer-Component-Bug Network used in [6]

2.2. For example, *Doc1* consists of terms - "crash", "ui" and "method". Having these documents, Lucene first pre-processes the terms of the documents, and constructs an inverted index as shown in Figure 2.7. Table 2.2 shows that the documents have six unique terms - "crash", "ui", "method", "file", "view" and button". So, the inveted index contains six entries to map the documents. The inverted index maps "ui" to all the three documents, as all the documents contain this term. Similarly, term "method" is mapped against *Doc1* and *Doc2* as they contain this term.

2.4.2.2 Graph Construction

While resolving bug reports a number of developers communicate with each other. A developer may have expertise of resolving bugs of different types. Again, a component may contain a huge number of bugs, than the other components. To leverage these information in ultimate fixer suggestions, various techniques constructs graph from the source and bug history. For example, Figure 2.8 shows a sample Developer-Component-Bug (DCB) which is used in [6] for extracting developer expertise information.

The directed DCB network of Figure 2.8 is generated using three types of

nodes, which are - *Developer*, *Component* and *Bug*. Here, the *Developer* node corresponds to actual fixers of the report. The *Component* node represents a set of source files changed by the developer and *Bug* represnts a fixed bug. The edges of the graph models the relationship between the developers and the source code components they worked on. The relationship between the components and the associated bugs represents which files were fixed during the bug resolution. These types of graphs are used for identifying the previous collaboration of developers. The relationship of the graphs can be leveraged in bug assignment to identify fixer teams.

2.4.3 New Bug Report Processing

When new bug reports arrive at the system, the bug assignment techniques tend to suggest developers with the knowledge learned using the previous two Subsections 2.4.1 and 2.4.2. Generally, the *summary* and *description* fields of the new report are leveraged for searching the constructed data ([8, 42]). However, techniques may also use other bug report fields such as *component*, *product* etc [6]. The new report keywords goes through the same preprocessing step mentioned earlier. This is required because the accuracy of the search depends on the relevancy of the query. A query is then formulated for searching the Lucene indexes. The search is performed using Lucene IndexSearcher. It first identifies the Directory where the constructed index is stored. Using the formulated query, the index is searched and a list of matching documents are returned.

In case of bug networks, the networks are parsed to identify communications among the developers. Based on the communications, developers are suggested

2.4.4 Ranking

Ranking is performed to identify the most relevant candidates from the search results. Various established information retrieval techniques are widely used while

identifying similar reports or developer profiles in bug assignment. Examples of such techniques include Term Frequency Inverse Document Frequency (Tf-idf) Weighting, Vector Space Model (VSM), Latent Dirichlet Allocation (LDA) etc. In the following subsections these techniques are introduced.

2.4.4.1 Term Frequency - Inverse Document Frequency

Term frequency - inverse document frequency, (tf-idf) technique is a statistical measure used to evaluate how important a word is to a document in a collection or corpus [43]. It is a widely used technique in text mining. The tf-idf weight of a term describes its importance in the overall collection. This importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

Let, t be a term in a document, d of a document collection having N documents. Then, $tf_{t,d}$ is referred as *term frequency* which measures the number of times t appears in d. While computing $tf_{t,d}$, all terms are considered equally important. However it is known that certain terms, such as "a", "is", may appear frequently but are of less importance. The rare terms of a document should get higher priority to represent its importance in the corpus. Thus, *inverse document frequency* is introduced and measured using following Equation 2.1 -

$$idf_t = log\frac{N}{df_t} \tag{2.1}$$

Where, idf_t refers to rareity of the term t in the corpus, and df_t denotes number of documents with term t, in the corpus.

Combining the definitions of *term frequency* and *inverse document frequency*, the tf-idf weighting scheme finally assigns a composite weight to term t in document d given by -

$$tf\text{–}idf_{t,d} = tf_{t,d} \times idf_t \tag{2.2}$$

Now, for a query q, composed of a number of keywords, the similarity between the query and document d, can be measured by summing up the *Tf-idf* weights of all the terms as follows -

$$Score(q, d) = \sum_{t \in q} tf\text{–}idf_{t,d} \tag{2.3}$$

The *Tf-idf* weighting scheme may be applied on to compare the new bug report with fixed bug reports and source history to identify expert developers.

2.4.4.2 Vector Space Model

The representation of a set of documents as vectors in a common vector space, is known as Vector Space Model (VSM). It is a fundamental technique used in document scoring, document classification and document clustering etc. VSM converts a query and document in vectors, and measures their similarity by calculating the cosine angle between the vectors. A higher score represents higher similarity between the vectors.

Let, $\vec{V}(q)$ and $\vec{V}(d)$ be the vector representation of the query q, and document d respectively. Then the cosine similarity score between q and d is calculated using following Equation 2.4 -

$$Score(q, d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)||\vec{V}(d)} \tag{2.4}$$

Here, $|\vec{V}(q)|$ and $|\vec{V}(d)$ denotes the magnitudes of the vectors. The cosine similarity measure has also been used for identifying similar vocabulary of developers by various techniques [1, 6]. Using this similarity between the new bug report and collected information sources, potential developers are derived.

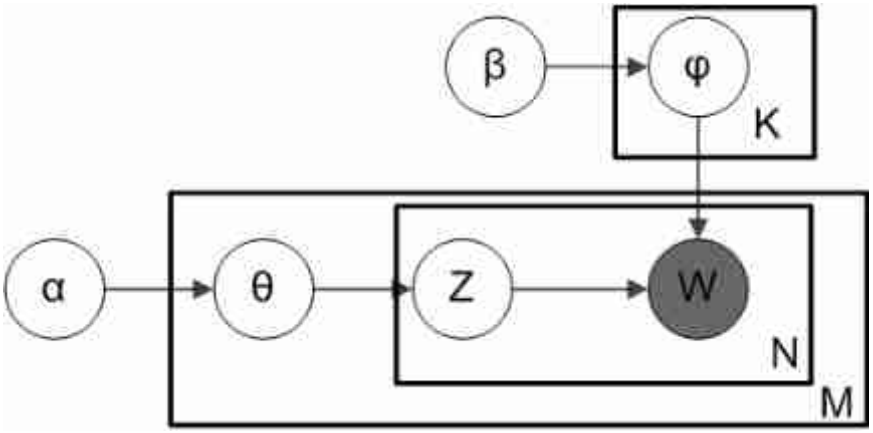

Figure 2.9: Graphical Representation of Latent Dirichlet Allocation Model

2.4.4.3 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) model is a statistical model intended to discover latent semantic topics in large collections of text documents [44]. The key insight into LDA is the premise that words indicate the topic of the document. Latent topics are thus discovered by identifying groups of words in the corpus, that frequently occur together within documents. Learning in this model is unsupervised because the input data provides only the words within documents, regardless of the topics in the documents.

A sample graphical representation of LDA is shown in Figure 2.9. LDA Starts processing with M documents having K latent topics. The number of words in a document is denoted as N as shown in Figure 2.9. LDA first randomly assigns a topic to each word of each document which is denoted as Z. It then gradually learns the topic of each word in the document, given the topic of the other variables and two prior variables α and β. Here, α denotes the initial probability of a topic in a document and β represents the probability of a word being to a corresponding topic. The main goal of LDA is to identify the two variables ϕ and θ. ϕ denotes a probability distribution where each entity denotes the probability of word belonging to a specific topic. Beside, θ represents another probability distribution where each entity denotes the probability of a topic to be contained

in the document. The topic for which a bug report gets the highest probability can determine the type of a report [13]. Besides, this score can also be used to measure similarity between new reports and existing information sources. A more detailed view of LDA is described in Subsection 5.2.2 of Chapter 5.

2.5 Summary

Automatic bug assignment has become an essential step for maintaining the quality of software systems. In this chapter, the basic essentials of bug assignment has been outlined for better understanding of the overall assignment procedure. A detailed description on software bugs, bug reports, bug reporting and maintaining practices are also presented. Besides this, a general architecture of bug assignment technique is explained which is composed of data collection, index and graph construction, new report processing, and ranking. The information flow of this chapter focused on demonstrating the basics of bug assignment for easy understanding. The next chapter briefly discusses the existing efforts in the filed of bug assignment.

Chapter 3

Literature Review

With emergence of technology, large scale software systems are being developed to meet user needs. The company owners are always in a hurry of product releases to hold customers and marketplace. Due to this tight time schedule, software developers generally get insufficient time for system designing, standard coding and product testing. As a result, software bugs arise. Software systems maintain software bugs using various Bug Tracking Systems (BTS), such as Bugzilla [16]. Studies have revealed that 50-60 bugs are reported in BTS on a daily basis. Therefore, for ensuring software quality, the reported bugs need to be resolved as soon as possible [45]. An appropriate developer who has expertise on the bug related field, and is currently working on the same field can fix the bug faster. Therefore, automatic and accurate bug assignment is an important task for faster bug resolution. Various techniques have been proposed by researchers to accurately delegate bugs to developers. In this chapter, a detailed study on the existing bug assignment techniques has been conducted. These techniques can be broadly divided into six categories - Text Categorization Based, Reassignment Based, Cost Aware Based, Bug Data Reduction Based, Source Based and Industry Oriented. Significant works related to each broad category are also described in this chapter.

3.1 Related Work

Concerned with the increased importance of automatic bug assignment, a number of techniques have been proposed by researchers. For recommendation, most of the existing techniques use previous fixes by developers. Again, some of the techniques consider current activities of developers for identifying actual fixers. A survey on various bug triaging techniques has been presented by Sawant et. al [46]. The survey divided existing bug triaging techniques into text categorization [6, 9–11], reassignment [39, 40, 47], cost aware algorithm [13], software data reduction [48–50] and source based techniques [1, 12] etc. Besides, studies focusing on industrial needs of bug assignment are also available in the literature [4, 14, 51]. Based on the focus and types of techniques considered in these approaches, the related studies are classified into six categories. These categories are pointed as follows:

- Text Categorization Based
- Reassignment Based
- Cost Aware Based
- Bug Data Reduction Based
- Source Based
- Industry Oriented

In the following sections, each of these approaches are described in detail.

3.2 Text Categorization Based Approaches

Text categorization based approaches are the most prevalent ones among the mentioned six types of assignment approaches. These approaches build a model

that trains from past bug reports [6, 9–11]. It then applies text similarity algorithms between the new and trained bug reports to predict correct rank of developers. The main target of these approaches is to recommend individual developers who have previous experience in solving similar bugs.

3.2.1 Bug Report Metadata

Most of the existing techniques formulate bug assignment as a classification problem, where instances represent bug reports, features represent textual descriptions and developers act as the class labels. However, the textual contents contain various noisy and irrelevant terms which lead these approaches to low prediction accuracy. Significant features depicting important characteristics of a reported bug can increase the prediction accuracy.

Banitaan et al. proposed TRAM, a bug assignment technique by using features other than the textual description to improve the prediction accuracy [9]. The approach first identifies the features for learning the model. The types of metadata associated with a bug report are taken as features - discriminating terms, bug component and reporter. The approach considers the title of a bug report for collecting the discriminating terms. Text preprocessing techniques such as white-space, punctuation and stop-word removal, stemming are applied on the title text. After that, a bug-term matrix is constructed and weighted by Term Frequency [51]. The discriminating terms are selected using Chi-Square term selection method. The predictive model is built using Naive Bayes Classifier for its better performance in terms of accuracy and computational efficiency [51].

TRAM was tested on four open source projects and its applicability was measured in terms of F-score. It was compared against two baseline approaches - one using all the terms of title as features and another using the discriminating terms of the title as features. The results were evaluated using two metrics - recall and precision. It shows that TRAM achieves higher precision and recall. The

results also discussed and justified the consideration of component and reported features using Gain Ratio.

The main limitation of TRAM is that, it assumes the assignee of the report as the only expert developer for fixing the report. This assumption is invalid, as bug fixing is usually a collaborative task. Therefore, team suggestion is an important requirement for bug assignment techniques. Again, TRAM only considers previous history and ignores recent activity of developers. As a result, new developers would not be assigned by the predictive model.

3.2.2 Fixing Time Meta Data

Previous efforts for creating automatic bug assignment systems use machine learning and information-retrieval techniques. One such commonly used information retrieval technique is tf-idf term weighting technique (already described in Chapter 2 Subsection 2.4.4.1). Tf-idf is a statistical computation technique for weighting terms based on their term frequency. However, tf-idf does not consider other metadata, when calculating the weight of the terms. One such metadata is the time metadata which refers to the time at which a term was used while fixing the reports.

Shokripour et al. proposed a Time aware Noun based Bug Assignment (TNBA) technique to improve the accuracy of fixer suggestion [2]. The proposed approach is divided into four steps - *Activity History Extraction*, *Entity Extraction*, *Term Weighting* and *Developer Selection*. The first step takes the fixed bug reports as input. The main task of this step is to identify the fixers of the report. Two techniques are used to link the reports to their associated fixers. Firstly, the attached source patches can be used to identify the fixers of the report. Secondly, in case of absence of patches, the commit history needs to be parsed to identify the link [52–54].

The *Entity Extraction* step determines the expertise of developers by examin-

ing the nouns found in the summary and description of fixed bug reports. It then constructs an index mapping the developers to used terms and term usage time. The *Term Weighting* step calculates the weight of a term from the perspective of each developer. The tf-idf technique is modified to add time metadata with it. That is, how important the term is for the developer and how recent the developer used the term. Finally, on arrival of new bug reports, the weight of the terms in the new report is calculated from the perspective of each developer. The highest scored developers are suggested as the list of potential fixers.

TNBA was applied on three open source projects for evaluation, which are - Eclipse, Netbeans and ArgoUML. The technique was compared with three state of the art techniques - tf-idf, VSM and Naive Bayes [55]. 200 test reports were randomly selected from each project to evaluate TNBA's accuracy of actual fixer assignment. The results show that TNBA improves the accuracy of the existing techniques.

Although TNBA considered the fixing time of the terms, it ignores the source commit recency of developers, which may lead to recommendation of inactive developers. The consideration of only past reports can not gain information about new developers. As a result, this technique would not assign tasks to new developers.

3.2.3 Developer-Component-Bug Network

Conventional bug assignment techniques use text categorization and machine learning approaches to automate this process. However, no significant study validates the effectiveness of these approaches on different project size. A study on 109 different Scrum teams revealed that the teams generally consist 4 to 18 members [56]. On the other hand, in large scale open source projects team size is large, even up to hundreds or thousands. The accuracy of conventional approaches may change with team sizes of developers. More study can identify

the effects of team size to propose generalized bug assignment approaches.

In order to propose a generalized bug assignment approach independent of developer team size, Hao et. al developed BugFixer, a developer recommendation method [6]. The proposed approach performs overall recommendation in four steps –bug report processing, bug similarity computing, constructing Developer-Component-Bug (DCB) network, and recommendation over DCB network. The bug report processing step extracts features(word) from bug summary and description and uses these features as inputs to machine learning algorithm to classify the bug reports. A tokenization algorithm is also proposed to process the composed keywords of bug reports. The algorithm first removes all the stop words from the keywords, transforms these into unified forms and splits these keywords according to the commonly-adopted naming conventions such as Java naming convention. This algorithm takes the keywords For computing similarity between new and existing bug reports Vector Space Model (VSM) is used. While computing similarity, the approach leverages source information besides bug report texts. The links between source code files and bug reports are established by parsing system source repositories. Next, a DCB network is built to determine the relevance between bug reports and developers. A DCB network is a directed graph where three kinds of nodes are considered - developer, component and bug. Two types of edges connect the nodes of DCB network. The first type connects the components to developers who worked on these components, and the second type connects the bugs to components to which those belong to. Finally, when a new bug arrives, recommendation is performed over the DCB network. The previously fixed bug reports are assigned weights based on their similarity with the new bug report using VSM. The approach then calculates the relevance between the new bug report and the developers level by level using defined equations. Based on the relevance value, BugFixer sorts the developers in a descending order and recommends developers for fixing the new bug.

BugFixer was evaluated on three open source projects and two industrial projects. The effectiveness was compared based on recall of the recommended list. The results show that BugFixer performs better than the conventional approaches for large team as well as project sizes. The result section also showed the comparison of recommendation accuracy with and without using DCB network.

As the DCB network is constructed using past bug reports, the recommendation list becomes less accurate with joining of new developer. The technique does not consider the number of issues currently assigned to existing developers. This will lead to over-specialization problem. The new developers would never be under consideration for bug assignment.

3.2.4 Developer Preference Elicitation

Besides considering developer's previous expertise, preference of developers towards solving a specific problem is another important factor. A study on Eclipse project reveals that 24% of the bugs are passed to another developer before final resolution [42]. If a developer is assigned a preferred task, s/he would get self-motivation and spend less time procrastinating. Keeping this question in mind, Baysal et al. [10] used Preference Elicitation technique to leverage developer feedbacks in the process of task assignment.

Baysal et al. have enhanced existing text categorization techniques by adding user preference of fixing certain bugs in recommendation process [10]. The framework performs its task using three components. The approach starts with the initial assumption that developers prefer to handle bugs in their area of expertise. So, the *Expertise Recommendation* component employs VSM to infer information about the expertise of developers from previously fixed bugs. It extracts keywords from the textual content of summary, description and comments to present the developer's profile. It then applies tf-Idf weighting scheme on the term vector to assign high weights to most relevant words. When new bug re-

port arrives, the term vector of developers and the incoming report are compared to make expertise recommendation of developers.

The *Preference Elicitation* component enables the developers to provide their willingness of solving certain bugs. A new field called "Rating" is attached to the bug report, to collect and store the preference level through a feedback process. A developer needs to label the bugs with one of three categories - "Preferred", "Neutral" and "Non-Preferred". The preference of a developer is gathered by creating a whitelist of bugs which the developer rated as preferred. The whitelist of a developer is a term vector, containing terms of the preferred bug reports. Now, for a incoming bug, it's similarity with the whitelists is measured to assign preference level to developers. Knowing the preference and expertise of each developer, the *Task Allocation* component combines these factors for fixer suggestion. While assigning tasks, the technique also considers current workload and availability of developers using a technique proposed by Weiss et al. [57].

The study did not present any experimental validation of the proposed approach. They justified this absence due to the complexity of designing the model, and unavailability of developer details. The approach basically depends on the preference given by developers. Developers may tend to fool the system by deliberately faking the preference rating. Another limitation of the proposed approach is new developers would get no tasks, as they have not fixed any bugs previously.

3.2.5 Code Authorship

As mentioned above, traditional approaches need to mine software repository for training recommendation models. Most of the techniques mine either of the two repositories - bug reports and source history (commits). Both of the techniques require extensive mining of huge information sources.

Concerned with the cost of extensive mining, Mario et al. came up with a different triaging approach [11]. This approach presented a code authorship

based incoming change request triaging approach. The proposed approach consists of two steps - locating relevant source portion for the incoming change (a bug or feature) and identifying authorship for task delegation. For locating the relevant source portion, the source code of the input project is indexed using Latent Semantic Indexing technique [58]. For indexing, only the identifiers and comments are extracted from the source files. As a result, each source file has a corresponding vector in the constructed index. When a new change request arrives, the long description of the request is extracted. It is then matched to the indexed files using LSI to obtain a ranked list of relevant source files.

The next step recommends a list of developers based on the identified source files mentioned in the previous paragraph. The source code files are converted into lightweight XML representation called, srcML [59]. The header comments are extracted via XML processing as header comments typically contain the copyright, licensing and authorship information. Regular expressions are devised to obtain authors from the textual header comments. Finally, the relevancy of an author is measured via calculating the frequency of appearance of his/her name in the comments of top n ranked source files. For tie breaking, the rank of n source files are considered.

The competency of the proposed approach was justified by its application on three open source projects. Its effectiveness was measured in terms of precision and recall and it successfully outperformed two benchmark systems.

The validity of the approach depends on the quality of comments provided by authors. However, source comments are vulnerable and they generally are not written following standard coding schemes. Again, an author may not reveal his information in the comments of the source files. Besides, a simple change in the comment may not be detected by the devised regular expressions. The approach ignores expertise information which can lead to assignment to inexperienced developers. If a developer has not yet commented on any source file, he would

not get any task in hand as well as will be deprived of future change requests.

3.3 Reassignment Based Approaches

When the assigned developer is unable or unwilling to solve a bug, s/he passes or tosses it to another developer for fixation, which is known as bug reassignment. Bug reassignment has negative impacts on overall maintenance process. The main reason behind bug reassignment is manual and error-prone assignment of tasks. Bug reassignment introduces cost and prolongs bug resolution time. Studies have reported that 37-44% bugs are tossed at least once, and one tossing event takes an average of 50 days [39]. Therefore, reducing these reassignments can improve the effectiveness of bug triaging systems.

3.3.1 Tossing Graph

A bug reassignment problem can be stated using a directed graph where the nodes represent system developers, and directed edges represent passing of a bug between two developers. A path in the graph depicts the passing history of a bug. The more the path length is reduced, the lesser reassignments occur. Using this trick, various tossing graph based bug triaging techniques have also been developed by researchers (that is [39, 40, 47]). The main focus of these techniques is to reduce the number of passes or tosses a bug report goes through because of incorrect developer assignment.

Jeong et al. invented bug tossing graphs to improve the accuracy of bug assignment techniques [39]. A tossing graph is generated from previous bug reports. While generating graphs, two types of models are introduced - actual model and goal oriented model. In actual model, every single tossing is visible in the graph. On the other hand, the goal oriented model is a decomposed version of the actual model connecting a developer to actual fixers only. All

the intermediate links between the first assignee and the fixer are encoded. In both the models, the edges are weighted according to the probability of a toss between two developers. Both the models are designed basing on the properties of Markov Chain model [60]. The paper also mentioned that in many cases, their model does not hold the Markov model properties.

Once a new bug report arrives, the technique applies traditional machine learning approaches on the indexed bug reports to retrieve the most similar ones. The developers of these similar bug reports are added into a list. Next, for each listed developer, the tossing graph is searched, and the developer who has strongest tossing relationship is returned. This returned developer is pushed at the following index of the enlisted developer. Finally, this updated list is predicted as ultimate fixers. To evaluate the approach, two machine learning algorithms were used - Naive Bayes and Bayesian Networks. The results show that the consideration of tossing information, improves the prediction accuracy. A limitation of this approach is search failures in the tossing graph. The paper reported that the reduction of less important tossing relationships introduces search failures in the graph.

An improvement of this study has been proposed by Chen et al. [47]. This approach leverages the same tossing graph used in [39]. This approach optimizes the tossing graph by deleting the retired or non-recent developers from the graph. When a new bug report arrives, VSM similarity technique is conducted to obtain the similar previous bugs. Then, the tossing graph is pruned using the retrieved similar bug reports. Now the list of developers found in the pruned sub-graph is recommended as potential developers. This study was evaluated using Mean Length of Tossing Paths (MLTP) and Failure Rate (FR) metrics. The results depict that the bug fixer is detected using fewer tossing events.

3.3.2 Multi Feature Incremental Learning

A fine grained incremental learning and multi feature tossing graph-based technique has been proposed by Bhattacharya et. al [40]. It improves the previous techniques by considering multiple bug report features like product and component while graph construction. The main contribution of the paper is introducing incremental learning after each new bug report recommendation. It updates the training set after each instance iteration for better learning of the model.

The tossing graph is generally constructed using previous bug reports. Therefore, as mentioned above, considering only historical activities lead to low accuracy of the recommended list. The main limitation of these approaches is that the tossing graph becomes less accurate with joining of new developers. As new developers do no have any bug fixing history, they would not be added to the graph. Thus, these approaches fail to meet the industrial need of equal resource utilization as well as team recommendation.

3.4 Cost Aware Based Approaches

CosTriage, a cost aware developer ranking algorithm has been developed by Park et. al [13]. The technique converts bug triaging into an optimization problem of accuracy and cost, which adopts Content Boosted Collaborative filtering (CBCF) for ranking developers. The motivation of the study was established by combining the two questions "Who can fix the bug?" and "Who can fix it faster or fix with lowest cost?"

The paper performs in two steps - constructing developer profile and ranking developers. A developer profile is a numeric vector that represents developers estimated cost for fixing certain types of bugs. In these context two questions need to be answered (1) how to determine the type of bugs and (2) how to obtain developer profile. For identifying bug types LDA technique is applied on

the bug repository. After determining bug types, the technique computes the average fix time for each bug type by a developer, to construct developer profile. If a developer has not previously fixed a certain type of bug, his profile contains missing values for those bug types. To fill up the missing values collaborative filtering technique is applied.

On arrival of a new bug report, the technique first identifies the type of the new bug report. It extracts words from title and description of the bug report and sums the word distribution for each topic. The highest scored topic is determined as the topic of new bug report. It then gets a score representing bug fixing cost, for each developer from the above mentioned developer profile. On the other hand, the technique applies content based recommendation technique on the new and existing bug reports to get developer experience. Each developer obtains an experience score based on the word similarity between new bug report and developers previously fixed bug reports. Finally, both these scores are summed and then ranked in a descending order to get the appropriate developer list.

If the technique fails to identify the type of the new bug report, it uses bug reports source code snippet to solve the problem. The technique takes 100 bug reports from bug repository which contains source code snippets. It then matches the source code import portions between the new and existing reports using Jaccard similarity coefficient. The type of the new bug report is then determined by the type of most similar bug report.

The proposed approach would fail if the new bug report does not contain source code snippet. Although the approach addressed the problem of new bug report, it did not provide any direction to handle new developers. The approach considers previous bug fixing rating which lack the information of new developers. Thus, this approach fails to be generalized in the industrial context.

3.5 Bug Data Reduction Based Approaches

Few data reduction techniques for effective bug triaging have also been proposed in literature (e.g. [48–50]). The main focus of these approaches is to remove noisy and duplicate data from the system's bug repository. Xuan et al. combined existing techniques of instance (such as duplicate. bug report) selection and feature (such as keywords) selection to simultaneously reduce the bug dimension and the word dimension [50]. The technique uses a feature selection and instance selection algorithm on existing bug repository. The reduced bug data contains fewer bug reports and fewer words than the original bug data and provide similar information over the original bug data.

The major flaw of data reduction based techniques is that the reduced data set is another representative of historical developer activity based information. Therefore, the prediction accuracy suffers due to inactive or retired developers. The reduced data set is not updated when a new developer joins. Thus, the new developers do not get any task for resolution.

3.6 Source Based Approaches

Few source based bug assignment techniques have also been proposed by researchers. Expertise models of developer can support task assignment. It has been proposed that task assignment can be improved if an externalized model of developer's expertise in terms of code commits is available [61]. Based on this finding, developer vocabulary based approaches have been derived by researchers. The focus of these approaches is to leverage source activities of developers.

3.6.1 Developer Vocabulary

Matter et. al has suggested Develect, a vocabulary based expertise model for recommending developers to assign bugs [12]. The approach initially parses the

source code and builds a model of the entire repository. Using *diff* command, then collects the word frequencies of changes between two versions of the same file. The identifier names and comments appearing on the changed files are considered as developer's vocabulary. Next, an expertise model is trained using existing vocabularies and stored in a matrix as a term-author-matrix. For new bug reports, the model checks the report keywords using lexical similarities against developer vocabularies. The highest scored developers are taken as fixers. For overcoming inactive developer recommendation, the technique uses a threshold value. The technique totally ignores experienced developers in recommendation process, which can lead to suggestion of novice or inexperienced developers. Besides, source comments are noisy information. In many cases developers may put irrelevant comments which would affect the approach validation. Along with this, the technique fails to model new developers as they may not have committed any changes yet. Subsequently, it increases bug reassignments, prolongs fixing time and reduces recommendation accuracy.

3.6.2 Commit Time Based

Most of the proposed approaches only considered previous commits of developers, ignoring the time of commits. The time meta data of committing a change can proved to be an important indicator of current activities of developers. It can also reduce the recommendation of inactive developers.

A time based bug assignment called ABA-Time-tf-idf technique has been proposed in [1]. The technique focuses on using time meta data, while identifying the recent developers for bug assignment. The technique first parses all the source code entities (such as name of class, method, method parameter and class attributes) and connects these entities with contributor to construct a corpus. In case of new bug reports, the keywords are searched in the index and given a weight based on frequent usage and time metadata. As a result, the more recent

a term is used by a developer, the more emphasis the developer gets. The high scored developers are shown at the top of the list.

Although ABA-Time-tf-idf correctly identifies recent developers, it generally fails to achieve high accuracy due to ignoring experienced developers. It also does not provide any support for industrial needs such as team recommendation and resource allocation by employing new developers.

3.7 Industry Oriented Approaches

Although open source and industrial projects have differences in terms of development process, requirements, size etc, both the systems express significant need for an effective bug triaging system. However, most of the existing approaches hypothesize and test the proposed triaging approaches focusing only on open source systems. Recent studies have pointed the industrial need of auto bug assignment ([14, 62]). The significant studies are outlined below.

3.7.1 Research-Industry Cooperation

In order to diminish the gap between current research and company needs, Vaclav et al. presented a study on their experience in dealing with automation of bug assignment within a research-industry cooperation [14]. The study was performed on one industrial project (a software company from Czech Republic) and another open source project (Firefox). The approach tends to test six devised hypotheses to compare the trend of bug assignment in the two fields. It uses Chi-Square and t-test for evaluating these hypotheses. The study revealed that the two considered data set were identical in terms of distribution. The study also identified that SVM+TF-IDF + stop word removal is the best classification model for both company-based and Firefox data. Finally, the paper concluded with some important findings which pioneer the future need of research in com-

pany based collaboration - the number of issues per developer is important before having reliable predication for ensuring task assignment to all individuals and the support for team recommendation.

3.7.2 Team Assignment

Industry always expects to obtain recommendation of developer teams for utilizing resources. The above mentioned studies clearly showed that considering heterogeneous features while training the model, improves recommendation accuracy. The connections among features can be viewed as a graphical network. An example of such network is - a network consisting of bug reports as nodes and developer names as edges (representing bug fixing). This network is refereed as homogeneous network as it contains only one type of node. The homogeneous network is incapable of identifying developer collaboration on bug resolution. It is so because, from this network, it cannot be identified which developers contributed together in a bug fixing process or on which component a set of developers work together. Concerned with the need of team recommendation, Zhang et al. developed KSAP, a bug report assignment technique using K Nearest Neighbour (KNN) search and heterogeneous proximity [4].

KSAP initially constructs a heterogeneous network using existing bug reports. Five types of entities are used as nodes in the heterogeneous network which are developer, bug, comment, component and product. These entities are connected with each other using ten types of relations such as - developer writes comments, bug contains comments etc. By parsing the fixed bug reports from repository, a heterogeneous network is built using those entities and relationships. For identifying developer collaboration, nine meta-paths are proposed in the paper. These meta-paths are categorized into three types, which are - collaboration on common bugs, components and products.

When a new bug report arrives, the technique converts the bug report into

a document vector. It then computes a cosine similarity between the document vectors of new and existing bug reports of repository. The K-nearest similar bug reports are taken as candidate list. Next, the developers who take part in the activities (such as toss, comment, fix, verify, reopen etc.) of these K similar bugs are also added into the candidate list. For each developer in the candidate list their associated meta-paths are extracted from the network. Finally, the technique computes a heterogeneous proximity score for each developer of the candidate list using those meta-paths. This score is calculated by summing how a developer collaborates with all the other developers in the candidate list, on common bugs, components and products. The candidates are ranked based on the heterogeneous proximity score, and top scored Q developers are recommended.

The proposed approach suffers from over-specialization problem due to ignoring latest activities of developers. Experienced developers will be queued with huge number of tasks. On the contrary, the new developers would not be able to take part in bug resolution.

3.8 Summary

The above detailed discussion on existing bug assignment techniques is an evidence that researchers have conducted significant efforts for automating this task. Six types of work regarding automatic bug assignment is visible in the literature - Text Categorization Based, Reassignment Based, Cost Aware Based, Bug Data Reduction Based, Source Based and Industry Oriented. However, most of the approaches are designed for solving triaging needs in open source systems. The existing solutions can recommend developers either using previous bug history or recent source commits. Considering only one of the information can not compensate the other which lead these approaches to inaccurate recommendation. Again, while assigning bugs, most of these approaches ignore important

development factors such as - team collaboration, equal task division and task allocation to new developers. Inconsideration of these factors makes the existing approaches non applicable in industrial projects. Therefore, considering and incorporating the above mentioned factors to provide a generalized assignment approach requires more research. In the next chapter, an expertise and recency based bug assignment approach has been developed. Using the idea from the next chapter, a team allocation technique has been devised later.

Chapter 4

Expertise and Recency Based Bug Assignment

Bug assignment is an essential step for fixing software bugs. A huge number of bug reports are reported daily in open source projects. Manually analysing and assigning bug report is error-prone and time consuming. Therefore, various automatic bug assignment techniques have been invented in the literature [26] [6], [1], [2] etc. Automatic bug report assignment is generally done using available information sources such as - bug reports, source code and commit logs. The bug reports which are fixed, indicate the expertise of developers in solving those bugs. However, with the passage of time, the developers are moved to different projects or company. On the other hand, the source code commits of software repository represents the current activities of developers. These recent activities cannot express developers expertise on any certain topic. So, considering only bug fixing expertise or recent commits may result in inactive or novice developer suggestion. It leads to reduced accuracy of the suggested developer lists. These limitations of the mentioned individual approaches raise the need for a combined approach to accurately assign bug reports to developers.

In this chapter, an automatic bug assignment approach is developed which

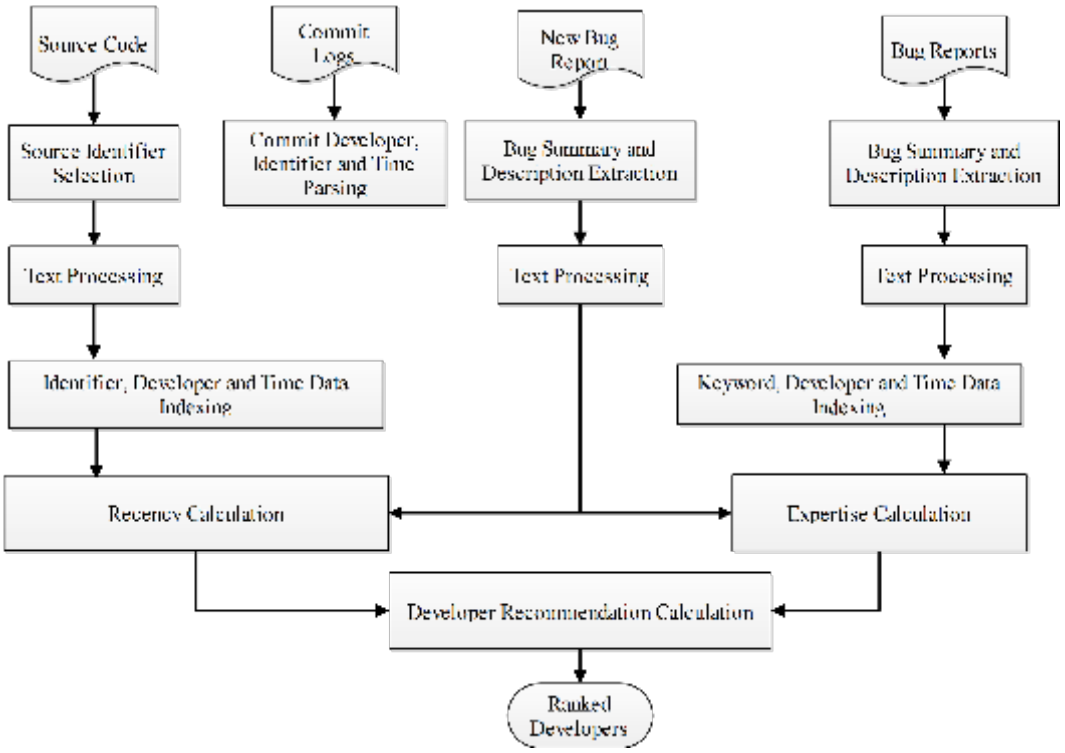

Figure 4.1: The Overview of ERBA

considers both the bug fixing expertise and source commit recency of developers. The approach is named as Expertise and Recency based Bug Assignment (ERBA). Here, functionalities of the modules of ERBA are illustrated in detail. Experimental results are also discussed for evaluating the applicability of ERBA. The applicability is measured using - Top N Rank, Effectiveness and Mean Reciprocal Rank (MRR) metrics.

4.1 Overview of ERBA

While designing the ERBA technique, attention is given to both developer's expertise and current activity based information. This is because, considering only one produces less accurate suggestions. ERBA performs developer suggestion using three modules, which are - *Recency Determination*, *Expertise Determination* and *Developer Suggestion*. The overview of ERBA is shown in Figure 4.1. The three processing modules of ERBA are pointed below -

- **Recency Determination:** Source commits of developers generally reveal their recent activity information. This module takes software source repository and commit logs as input. It then extracts the source code entities from the source files. The entities extracted from the source classes include the name of the classes, methods, method attributes, method parameters and package declarations. This module goes through the input commit logs to identify the change history regarding each of these entities. It then extracts the developer name and date associated with the committed entities. Lastly, this module constructs an index, mapping each source entity against the developers who committed the entities.

- **Expertise Determination:** The *Expertise Determination* module receives bug reports as input in XML format. The fixed bug reports indicate developers' experience on solving certain bug reports. So this module extracts the *summary* and *description* property of each bug report along with the fixer information from the bug report. It then generates another index to represent the expertise information of the developers. Each entry of the index maps a bug report keyword to the developers who previously fixed this keyword related bug report.

- **Developer Suggestion:** This module performs developer rank suggestion on arrival of new bug reports. When a bug report arrives at the system, it extracts the *summary* and *description* of the bug report. It then formulates a query using the extracted term from the bug report. This query is searched in the both indexes constructed in the previous two steps. The tf-idf technique is then applied on the search results to calculate a score for each developer. The score is called an ERBA score. Finally, top ERBA scored developers are suggested as potential bug fixers.

In the next sections, the detailed processing of these modules are presented.

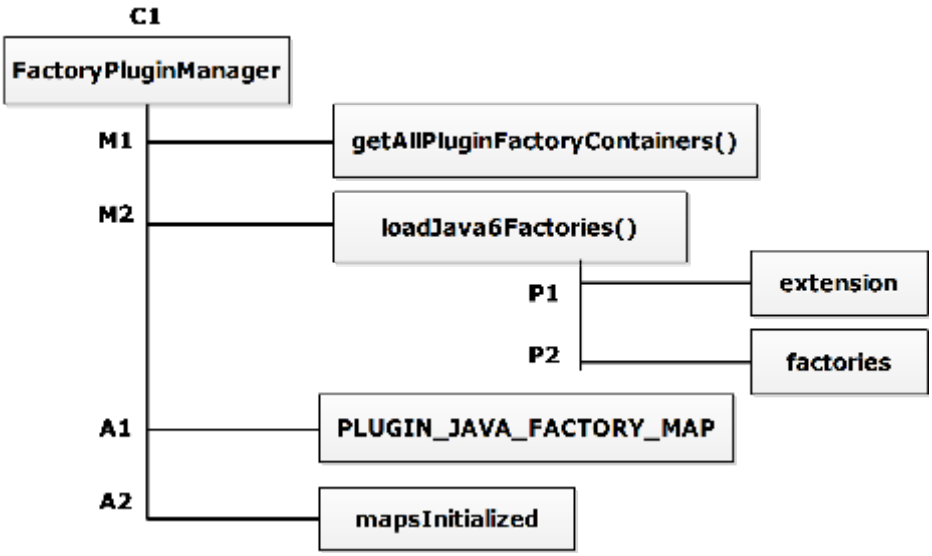

Figure 4.2: Partial Structure of Class *FactoryPluginManager* of Eclipse JDT

4.2 Recency Determination

Determining developers' recent activities while assigning bugs, helps to eliminate inactive developer considerations. So, the *Recency Determination* module of ERBA is responsible for identifying developers' recent activities. This information is generally found in their source code commits. So this module takes software source code and VCS commit logs as inputs as shown in Figure 4.1. It then parses each of the source files in the repository to extract the source entities. The names of classes, methods, attributes and method parameters are extracted from the source files. These data are taken into consideration as the source code entities represent developer vocabularies and activities.

Moreover, data by nature may contain unnecessary characters and keywords. For improved tokenization, all these parsed data undergoes through text processing. The text processing step contains identifier decomposition based on CamelCase letter and separator character, number and special character removal, stop word removal and stemming. Each of these text processing steps are described in Chapter 2 Subsection 2.4.1. For retrieving relevant results, both the complete and decomposed identifiers are considered for indexing. A sample of the source

```
<commit>
    <repo>eclipse.jdt.ui</repo>
    <hash>8cf0ece7695dbbe9bbebfcd3d1f44b601b205619</hash>
    <author>Dani Megert</author>
    <authorMail>dmegert</authorMail>
    <date>Thu, 8 Dec 2011 12:36:30 +0100</date>
    <subject>Fixed bug 365994: [preferences] Null analysis
    <note>Dani Megert</note>
    <changedFiles>
        <filePath>PreferencesMessages.properties</filePath>
    </changedFiles>
</commit>
<commit>
    <repo>eclipse.jdt.ui</repo>
    <hash>b441737900fbbd3fe2346d6c05daa02ec061f6c1</hash>
    <author>Markus Keller</author>
    <authorMail>mkeller</authorMail>
    <date>Mon, 5 Dec 2011 16:46:53 +0100</date>
    <subject>Bug 357325: [render] Method parameter annotati
    <note>Markus Keller</note>
    <changedFiles>
        <filePath>FactoryPluginManager</filePath>
        <filePath>JavaElementLabelsTest</filePath>
        <filePath>ASTNodes</filePath>
        <filePath>JavadocView</filePath>
```

Figure 4.3: Partial Commit Log of Eclipse JDT

entities of class *FactoryPluginManager* of Eclipse JDT is shown in Figure 4.2. The Figure 4.2 shows that this class has two methods named - *getAllPluginFactoryContainers* and *loadJava6Factories*. This class has two attributes as well, which are *PLUGIN_JAVA_FACTORY_MAP* and *mapsInitialized*. Here, *extension* and *factories* are two parameters of method *loadJava6Factories*.

The commit history of the project is also extracted in XML format using git commands. A sample of the extracted commit history of Eclipse JDT is illustrated in Figure 4.3. It is clearly seen that the commit XML contains a number of attributes such as - commit id (hash), author, commit date, subject and a list of changed identifiers associated with the commit. For example

- the second commit of Figure 4.3 shows that developer "Markus Keller" has committed changes to four class identifiers such as - "FactoryPluginManager", "JavaElementLabelsTest", "ASTNodes" and "JavadocView". It also shows that the commit is performed at *5 Dec, 2011*.

In order to determine developers' recency of using identifiers, proper links between the extracted source identifiers and commit logs need to be established. Again an identifier can be used by a number of developers in different phases of time during the project development. For searching recent usage of the identifiers by the developers, an index of all the identifiers needs to be built. So, following Algorithm 4.1 is proposed which performs the task of developers' source related activity collection.

Algorithm 4.1 Recent Activity Collection

Input: A list of string identifiers (I)
Output: An index associating identifiers with a postinglist of developers, ($DeveloperActivity$). Each $DeveloperActivity$ represents a data structure containing developer name ($name$) and identifier usage date ($date$)

```
1: Begin
2: Map <String,List <DeveloperActivity>>postingList
3: DeveloperActivity d
4: for each i ∈ I do
5:     Commits ← getCommitsOfIdentifier(i)
6:     for each c ∈ Commits do
7:         d ← new DeveloperActivity()
8:         d.name ← c.author
9:         d.date ← c.date
10:        if !postingList.keys.conatins(i) then
11:            postingList.keys.add((i)
12:        end if
13:        postingList[i].developers.add(d)
14:    end for
15: end for
16: End
```

Algorithm 4.1 takes a processed list of source identifiers as input. In order to associate the list of identifiers with corresponding developer list, a complex data structure, *DeveloperActivity* is constructed. It contains two properties -

developer *name* and commit *date*. To construct the index, initially an empty *postingList* is declared (Algorithm 4.1, line 2). An empty variable, d of type *DeveloperActivity* is also declared in line 3 for populating the *postingList*. A *for* loop is defined to iterate through the identifier list, I for index construction (Algorithm 4.1, line 4). For each identifier, i corresponding commits in which the identifier is changed, are extracted by calling function *getCommitsOfIdentifier* as shown in line 5. This function takes an identifier as input and returns a commit list of type *Commit*. Each *Commit* contains aforementioned commit log attributes. A nested inner loop is also defined to iterate on the *Commits* list (Algorithm 4.1, line 6). Each iteration of this loop initializes d with a new *DeveloperActivity* instance. It then updates the *name* and *date* property of d with *author* and *date* property of commit, c respectively (Algorithm 4.1, lines 7-9). For adding identifiers into the *postingList*, the list is first checked whether it already contains the identifier (Algorithm 1 line 10-12). In the next step, the updated value of d is added to the *postingList* against the identifier (Algorithm 4.1, line 13). This *postingList* will be searched later using identifiers found in the arrived bug reports. Finally, Algorithm 4.1 returns an index of triplet data structure containing identifiers, developers who use those identifiers and their identifier using date.

Continuing with the example of Figure 4.2 shows that class name, *FactoryPluginManager* is first processed into *factory*, *plugin* and *manager*. Besides, the second commit of Figure 4.3 represents that the identifier, *FactoryPluginManager* is changed by developer, "Markus Keller". So, Algorithm 4.1 maps *name*, "Markus Keller" and *date*, "5 Dec, 2011 16:46:53" against the identifier *factory* in the constructed index.

4.3 Expertise Determination

The frequency of fixed bugs by a developer is another key indicator for identifying appropriate bug fixers. The higher number of times a developer fixed certain keyword related bugs, the higher potential the developer has to solve those keyword related bugs. The *Expertise Determination* module of Figure 4.1 performs the task of identifying expert developers, who are proficient for solving similar bugs. A fixed bug report contains the full resolution history of the bug report. For this purpose, the bug reports are collected from bug tracking systems (such as Bugzilla). The *Expertise Determination* module then takes those bug reports for identifying developers' experience information. These bug reports are input in XML format for making it program readable. A partial XML structure of Eclipse bug #264606 is shown in Figure 4.4. It shows that the bug report contains a number of attributes such as *id*, *developer*, *creation_time*, *summary*, *description*, etc.

The bug repository contains either of two types of bugs - open and closed. As for identifying expert developers, the technique requires to index bug reports which are already fixed by developers. Therefore, the proposed technique takes closed bug reports into consideration. A closed bug report can have *resolved* and *verified* as bug *status*, and *fixed*, *invalid*, *wontfix*, *duplicate* and *worksforme* as bug *resolution*. The detailed specification about these attributes are enlisted in Table 5.1 of Chapter 5. So, the bug reports which are *resolved* and *verified*, and its resolution is *fixed* by developers, are used by ERBA. Again, the *summary* and *description* properties of a bug report represents details about the features of the bug. So, these property values are used by ERBA.

As, a term related bugs can be fixed by several developers, an index is required which connects the terms with these list of developers. For this purpose, the Algorithm 4.2 is constructed. The algorithm is input with a list of fixed

```
<bug>
    <id>264606</id>
    <developer>Thomas ten Cate</developer><developer_username>ttencate
    <creation_time>2009-02-11 17:30:00 -0500</creation_time>
    <product>JDT</product>
    <component>UI</component>
    <bug_severity>normal</bug_severity>
    <summary>[extract method] extracting return value results
    in compile error</summary>
    <description>Build ID: I20080617-2000 Use Extract Method on 'foo'
        class Bar {
            boolean foo;
            private boolean getFoo()
            {
                return foo; // use 'Extract Method' on 'foo' here
            }
            private void setFoo(boolean newFoo)
            {
                this.foo = newFoo; // breaks this line
            }
            this.bar() = foo;}
    </description>
    <comment>
    <comment_id>1553734</comment_id><who>Markus Keller</who>
```

Figure 4.4: Partial XML Formatted Bug Report of Eclipse JDT

bug reports (B). Each member of B has three properties - a list of bug report keywords (*terms*), the name of a developer who fixed the bug (*name*) and bug fixing *date*. The *terms* list is constructed using the keywords of bug *summary* and *description*. All the members of *terms* go through the same text processing step mentioned earlier.

Algorithm 4.2 Expertise Collection

Input: A list of bug reports (B). Each (B) is a complex data structure containing two properties - report keywords (*terms*), bug fixer name (*name*) and bug fixing date (*date*)

Output: An index connecting the bug report keywords with a postinglist of developers of type *DeveloperExpertise*. An instance of *DeveloperExpertise* is composed of two properties - *name* and fixing *date*.

```
1:  Begin
2:  Map < String, List < DeveloperExpertise >>
        postingList
3:  for each b ∈ B do
4:      DeveloperExpertise d ← new DeveloperExpertise()
5:      d.name ← b.name
6:      d.date ← b.date
7:      for each term ∈ b.terms do
8:          if !postingList.keys.conatins(term) then
9:              postingList.keys.add(term)
10:         end if
11:         postingList[term].add(d)
12:     end for
13: end for
14: End
```

Initially for storing the mapping between bug report terms and bug fixers, a *postingList* is declared (Algorithm 4.2, line 2). Each entry of this list connects a *term* to a list of complex data structure, *DeveloperExpertise*. For iterating through the bug reports (B) an outer loop begins at line 3. On each iteration, a new instance, d of type *DeveloperExpertise* is initialized (line 4). This instance is then populated with the fixer *name* and fixing *date* (line 5-6). An inner for loop is then declared for iterating on the *terms* list of each B (Algorithm 4.2, line 7). Before adding each *term* in the *postingList*, the list is checked for the term's existence in the list (Algorithm 4.2, line 8-10). Finally, the *term* mapping to a

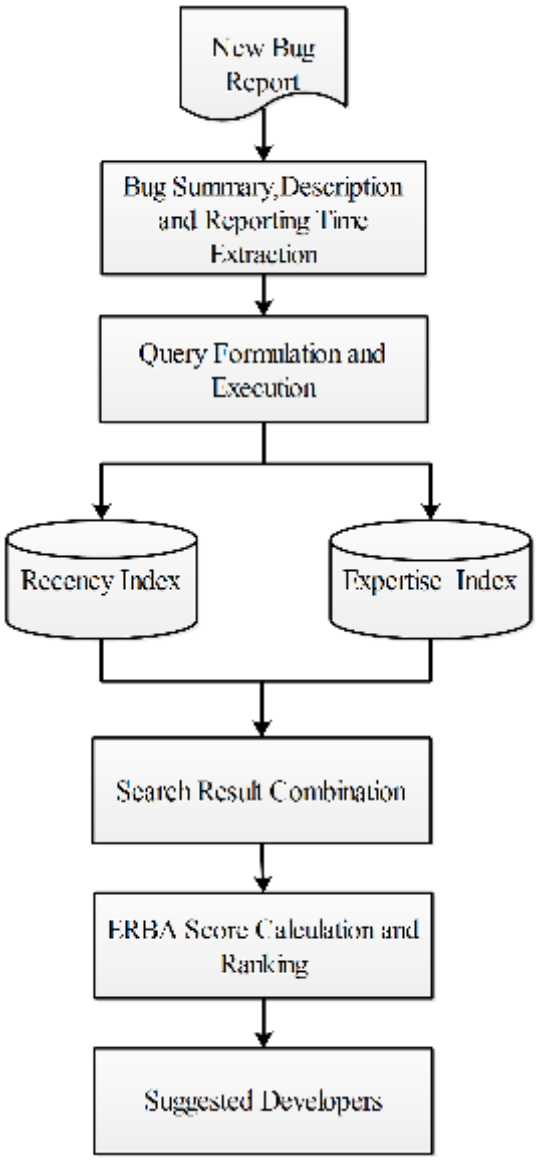

Figure 4.5: Developer Suggestion Procedure of ERBA

developer information, d is added to the *postingList* ((Algorithm 4.2, line 11)). As a result, another index containing the expertise information of the developers is constructed.

4.4 Developer Suggestion

Finally, ERBA combines both the recency and expertise information gained from above mentioned indexes for appropriate developer ranking. The overall devel-

Table 4.1:
Explanation of Attributes and Methods of Algorithm 4.3

Variables	
useFreq	no. of times a term is used in source commits by a developer
useDate	last usage time of a term by a developer
fixFreq	no. of times a term related bug is fixed by a developer
fixDate	last fixing time of a term by a developer
#dev	no. of developers in the project
Methods	
devUseFreq(t)	takes term, t as input and returns the no. of developers who commit the term
devFixFreq(t)	takes term, t as input and returns the no. of developers who fixed the term related bugs

oper suggestion procedure is represented in Figure 4.5. When a new bug report arrives, the *Developer Suggestion* module first extracts the *summary*, *description* and *reporting date* of the report. A search query is formulated using the keywords of the *summary* and *description* field of the report as shown in Figure 4.5. This query is then executed on the two indexes built by the *Recency* and *Expertise Determination* modules. Both the queries return a set of developers who uses or fixes the term of the new bug report. This module then combines these query results and construct a complex data structure of type *Map<String,Map<String,TermInfo>>*. The outer map associates each developer with a list of new bug report terms used by this developer. On the other hand, the inner map connects each term, to its usage information of type *TermInfo*. *TermInfo* represents a data structure consisting of four properties - *useFreq*, *fixFreq* , *useDate* and *fixDate*. The explanation of these properties is shown in Table 4.1.

Next, the combined search results are used to calculate an expertise and recency score for each developer. This score is called an ERBA score. This score calculation task is performed using Algorithm 4.3. It takes a new bug report (B) as input. The *getDeveloperTermUsageInformation* function takes B as input, and performs the above mentioned search query formulation and execution

Algorithm 4.3 Developer Suggestion

Input: A new bug report (B). (B) contains a set of keywords ($terms$) and bug reporting date ($Date$).
Output: A sorted developer list ($devList$).

```
1: Begin
2: Map < String, Map < String, TermInfo >>
       devTermInfo ←
           getDeveloperTermUsageInformation(B)
3: List < Developer > devList
4: for d ∈ devTermInfo do
5:     double recency, expertise
6:     Date fixDate, useDate
7:     for term ∈ devTermInfo[d.name] do
8:         t ← devTermInfo[d.name][term]
9:         tfIdf ← t.useFreq * log(#dev/devUseFreq(t))
10:        useDate ← (1/devUseFreq(t))+ (1/√(B.Date − t.useDate))
11:        recency+ ← tfIdf * useDate
12:        tfIdf ← t.fixFreq * log(#dev/devFixFreq(t))
13:        fixDate ← (1/devFixFreq(t))+ (1/√(B.Date − t.fixDate))
14:        expertise+ ← tfIdf * fixDate
15:    end for
16:    dev = newDeveloper()
17:    dev.name ← d.name
18:    dev.score ← recency + expertise
19:    devList.add(dev)
20: end for
21: devList.sort()
22: End
```

task (Algorithm 4.3, line 2). As a result, it returns complex data structure *devTermInfo* constructed from query results. An empty list of developers, *devList* of type *Developer* is declared in line 3 for storing ultimate developer rank. An outer loop is defined at line 4 to iterate on *devTermInfo*. Two empty variables *recency* and *expertise* are declared for storing each developer information, d (line 5). The variables *fixDate* and *useDate* are initialized at line 6 for getting the time information associated with the terms. Next, an inner loop is constructed to iterate over the new bug report terms used by this developer (line 7). For each *term*, its corresponding term usage information t of type *TermInfo* is extracted from *devTermInfo* (line 8).

For accurate developer recommendation, the recent activities is an important factor. Tf-idf term weighting technique is applied for measuring the frequent use of a term by a developer (line 9). This technique determines the weight of a term using the frequency of its usage ($useFreq$) and the generality of it in the project ($log(\#dev/devUseFreq(t))$). Considering only frequent use of developer may result in recommendation of inactive developers. To alleviate this problem, the recent use of a term is incorporated with its frequent use. The recency of a term is determined by adding the inverse of the number of developers who used this term, and the inverse of date difference between the bug reporting date ($Date$) and this term using date ($useDate$) (line 10). The greater the value of *devUseFreq(t)* is, the more important the term is for the developer. Again the greater the date interval ($B.Date–t.Date$) is, the less recent the term is used. This recency of a terms ($date$) is then multiplied by its frequency (*tf-idf*) to calculate developer's recency on this term. Finally, the sum of recency of all terms determines the developer recency (line 11).

On the other hand, the expertise of a developer in each term is calculated in similar tf-idf technique using term fixing frequency (*fixFreq*) and the generality of fixing the term related bugs (*log(#dev/devFixFreq(t))*) (line 12). The developer who fixed a term recently should get higher priority than the developer who fixed it a long time ago. Hence, the recent fixing of a term is incorporated with the tf-idf weight in a similar manner as shown in line 13. Developer's expertise is determined by summing the expertise of fixing all terms (line 14). Lastly for each iteration of *devTermInfo*, an instance, *dev* of type *Developer* is constructed, initialized (using developer name and ERBA score) and inserted into *devList* (line 16-19). The technique incorporates the recency and expertise value for assigning ERBA score to developers. This value represents both the source code and bug fixing related expertise and recency of the developers. As a result, Algorithm 4.3 ends its task by sorting the *devList* in a descending order based on ERBA

scores. ERBA concludes its task by suggesting the developers at the top of the list as appropriate developers for fixing the new bug.

4.5 Result Analysis

For evaluating the compatibility of ERBA, experimental analysis have been conducted on three projects. The projects are - Eclipse JDT [63], AspectJ [64] and SWT [65]. These projects are commonly used in evaluating bug assignment techniques ([1, 4, 13]). Moreover, the source repository, commit logs and bug repository of these projects are available in open source. The experimental analysis is focused to evaluate the ranking of the actual fixers for the new bug report assignment. The suggested developer list by ERBA is evaluated using three metrics - Top N Rank, Effectiveness and Mean Reciprocal Rank (MRR). The results of ERBA is compared with one source commit recency based and one previous bug history based technique known as ABA-time-tf-idf [1] and TNBA [2] respectively. The details of the experimental arrangements are demonstrated in the following Subsections.

4.5.1 Dataset Collection and Preparation

As ERBA takes the source repository, commit logs and bug reports as input, these information are collected for each of the three studied subjects. The properties of the collected data are enlisted in Table 4.2. The details of these datasets are explained below.

- **Eclipse JDT:** JDT stands for Java Development Tools, which is a subproject of Eclipse [63]. It supports the implementation of a Java IDE, for developing java applications as well as Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench. It also includes a Java compiler, incremental builder, number of views, editors,

Table 4.2: Properties of Experimetal Dataset

Project	**No. of Commits**	**No. of Java Classes**	**No. of Bug Reports**	**No. of Developers**
Eclipse JDT	25,700	6899	2500	898
AspectJ	7,736	3044	1522	378
SWT	24,265	1981	2500	670

wizards, and code merging and refactoring tools. The JDT project is composed of five components which are - APT, Core, Debug, Text and User Interface (UI). The source repository is maintained using git version control system. So, the source code and full commit history of JDT is collected using git commands. The collected JDT source repository contained 6899 java classes as shown in Table 4.2. Total 25700 commits are performed by developers, which are also gathered. Besides 2500 bug reports having *fixed* as bug *status* are extracted for training and testing purpose of the experimental analysis. All the bug reports are extracted in XML format as shown in Figure 4.4.

- **AspectJ:** AspectJ is an Aspect-Oriented Programming (AOP) extension for the Java programming language [64]. The motivation for AspectJ is the realization that there are functionalities which are not well supported by traditional programming methodologies. The added functionalities of AspectJ include error checking and handling, synchronization, context-sensitive behaviour, performance optimizations, monitoring and logging, debugging support, and multi-object protocols. The source repository of AspectJ is cloned using Git command line arguments. The commit logs are also extracted in XML format as shown in Figure 4.3. Total 3044 java files and 7736 commits are assembled as enlisted in Table 4.2. For identifying the expertise information of developers, 1522 *fixed* bug reports are used.

- **SWT:** The Standard Widget Toolkit (SWT) is a graphical widget toolkit

for use with the Java platform [65]. SWT is written in Java and is an alternative to the well known toolkits such as - Abstract Window Toolkit (AWT) and Swing Java GUI toolkit. It provides a portable API and tight integration facilities with underlying OS. This allows the toolkit to immediately reflect any changes in the underlying OS while maintaining a consistent programming model on all platforms. In case of SWT, the source repository contained 1981 java files, which are collected from SWT's git repository. During developing the project, the developers performed 24265 commits, which are collected as well. Finally total 2500 *fixed* bug reports are selected randomly for learning purpose of ERBA.

A set of bug reports are selected randomly from each of the studied projects for testing purpose. The suggested developer list for each of the test reports are then checked against the actual fixers of the report. The actual fixers are extracted from the bug reports as mentioned in Chapter 2, Subsection 2.2.2.

4.5.2 Evaluation Metrics

For measuring the performance of ERBA three metrics have been used. These are - Top N Rank, Effectiveness and MRR. All these metrics are widely used for evaluating the suggested developer list of bug assignment techniques. Top N Rank checks whether the N suggested developers contain the actual fixers. In this case, the higher the value of the metric, the better the accuracy of the approach. MRR evaluates the correctness of the first relevant developer. A higher value of this metric represents higher compatibility. On the contrary, Effectiveness checks the compatibility of ERBA in reducing false positives. The less number of false positives are considered, the more effective ERBA is. MRR refers The details of these metrics are as follows -

- **Top N Rank or Accuracy:** Top N rank or accuracy refers whether the

top N suggested developers contain the actual bug fixer, where *N=1,3,5,...,n* [1]. If the top N developers contain any of the actual fixers, the suggestion is considered correct. For example, N=3 refers, an actual fixer is obtained in the top 3 suggested developers. A higher value of this metric represents higher accuracy of ERBA.

- **Effectiveness:** In bug assignment, effectiveness is measured using the position of the first relevant developer in the ranking. Approaches that recommend relevant developers at the top of the list are considered more effective. It is so because ranking relevant developers at the top reduces the false positives which a triager needs to consider while bug assignment. In this context, the lower the value of the metric, the less effort is conducted by the triager, leading to more effective bug assignment.

- **Mean Reciprocal Rank:** The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct suggestion. The Mean Reciprocal Rank (MRR) is the average of the reciprocal ranks of results for the test queries. It is basically the inverse of the effectiveness metric. The MRR value is measured using Equation 4.1 similar to [1] -

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{Rank_i} \tag{4.1}$$

  Where, Q refers the number of test reports and $Rank_i$ refers the rank position of the first correct developer for the ith test report.

4.5.3 Research Question and Evaluation

The evaluation of ERBA is performed by achieving two goals which answers the first research question of Chapter 1, Section 1.3. These goals are titled as - **Goal1** and **Goal2**. **Goal1** analyses whether the actual fixers are achieved at

Table 4.3: Performance of ERBA on Three Studied Projects

Project	**No. of Test Report**	**Top 1**	**Top 3**	**Top 5**	**Top 10**	**Top 15**	**Avg. Effecti -veness**	**MRR %**
Eclipse JDT	500	127 (30.8%)	233 (56.4%)	309 (74.8%)	317 (90.3%)	400 (96.85%)	2.05	48.65
AspectJ	322	65 (20.5%)	167 (52.7%)	253 (79.8%)	310 (97.8%)	316 (99.68%)	2.02	43
SWT	514	104 (22.7%)	189 (41.2%)	321 (69.9%)	427 (93%)	450 (98%)	2.5	40.04

the top of the suggested lists by ERBA. Besdies, **Goal2** demonstrates how the combination of recency and expertise increase the accuracy of the suggested list. The following discussion demonstrates how ERBA achieves these goals.

Goal1: How many actual fixers are successfully suggested at the top of the list?

For achieving this goal, ERBA is applied on the three open source projects of Table 4.2. For each of the projects, a number of test reports are selected randomly as new bug reports. The suggested list of each of those test reports are then checked against the actual fixers of the test reports. This analysis is evaluated using the above mentioned three evaluation metrics. The results of the experimental analysis are reported below.

For JDT, 500 bug reports are selected randomly for testing purpose as shown in Table 4.3. The number of times actual fixers are obtained at the top 1, top 3, top 5, top 10 and top 15 are calculated. For example, 30.8% cases ERBA shows the correct developer at position 1. That is, 29% cases the first developer may correctly fix the bug without tossing it to other developers. Moreover, 56.4%, 74.8%, 90.3% and 96.85% cases the actual fixers are obtained at the top 3, top 5, top 10 and top 15 ranks respectively. These, higher values indicate that ERBA can successfully place the actual fixers at top of the suggested list.

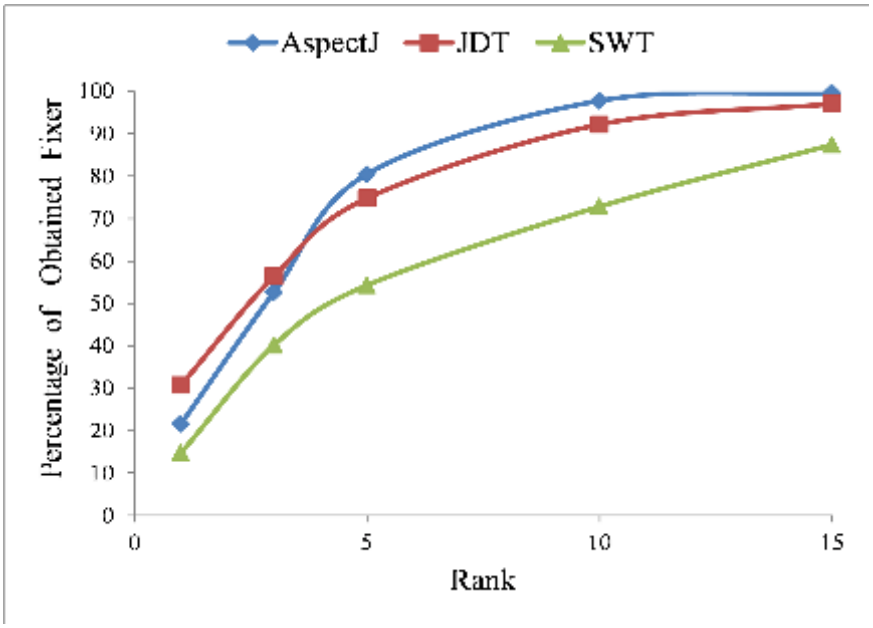

Figure 4.6: Top N Ranking of ERBA on Three Studied Projects

In case of, AspectJ 322 bug reports are applied on ERBA. Table 4.3 shows that 79.8% cases the actual fixers are found within top 5 position, which is also promising. It also represents that for AspectJ 99.68% of the actual fixers are successfully obtained by ERBA. 514 test reports are used for testing on SWT. Table 4.3 shows that for SWT, 22.7%, 41.2%, 69.9%, 93% and 98% actual fixers are retrieved at top 1, top 3, top 5, top 10 and top 15 ranks respectively. For all the projects, average 70% of the fixers can be suggested within position 3 which is significant for a bug assignment technique. The higher value of this metric for all the projects indicates higher applicability of ERBA.

In order to understand the ranking growth of ERBA for the three studied subjects, the percentage of fixers obtained at each N Rank is plotted in Figure 4.6. With the increasing number of rank position, a linear growth in the percentage of fixers obtained is found. Finding such a linear growth is expected, because an increment in the rank increases the probability of obtaining the actual fixer. Figure 4.6 also depicts that for all the three projects above 95% of the fixers can be suggested successfully by ERBA.

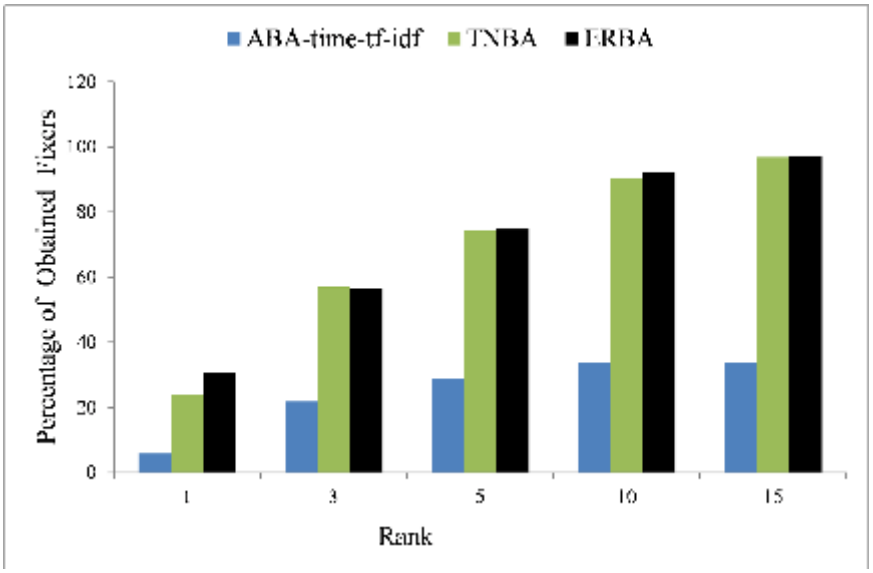

Figure 4.7: Comparison of Top N Ranking on Eclipse JDT among ABA-time-tf-idf [1], TNBA [2] and ERBA

The average effectiveness and MRR achieved by ERBA on the three projects are also enlisted in Table 4.3. If a technique shows the actual fixer at the top of the list, the bug triager does not need to go through the whole list. Thus, the lower the rank of the relevant fixers is, the higher effective the technique is. It is shown in Table 4.3 that for JDT, ERBA suggests the first actual fixer on average near rank 2.05 which is quite low . It indicates that the triager finds the actual fixer within the top 2 ranks of the list. Thus, it reduces the false positive a triager needs to handle. For AspectJ and SWT, obtained average effectiveness is 2.02 and 2.5, which are also pomising. Again, a higher value of MRR implies higher accuracy of bug assignment. The MRR values for the JDT, AspectJ and SWT are 48.62, 43 and 40.04 respectively. The lower value of the average effectiveness and higher value of MRR proves ERBAs applicability in bug assignment.

Goal2: Does the consideration of expertise information along with recency improves the accuracy of bug assignment?

Consideration of only recent commits may result in bug assignment to novice developers. So, ERBA considers both the expertise and recency information of developers for assigning newly arrived bug reports. To achieve this goal, the per-

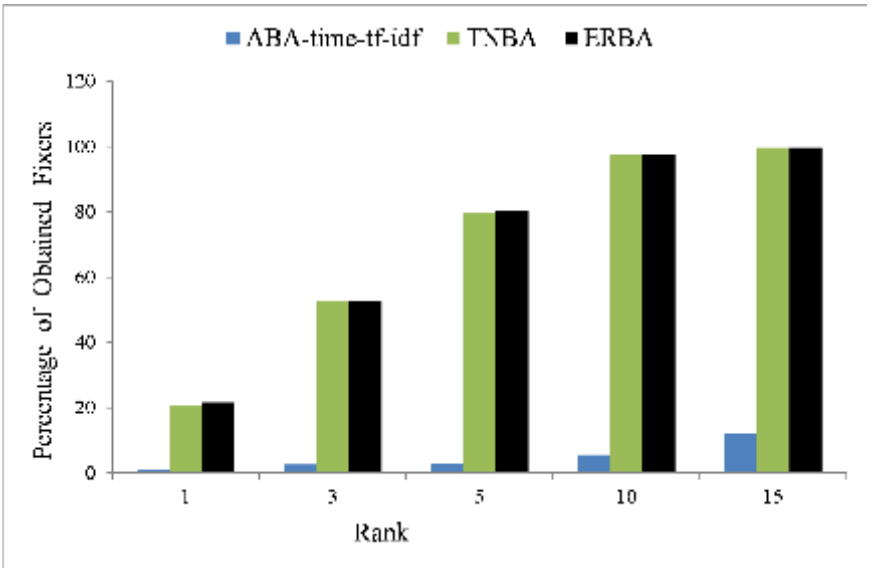

Figure 4.8: Comparison of Top N Ranking on AspectJ among ABA-time-tf-idf [1], TNBA [2] and ERBA

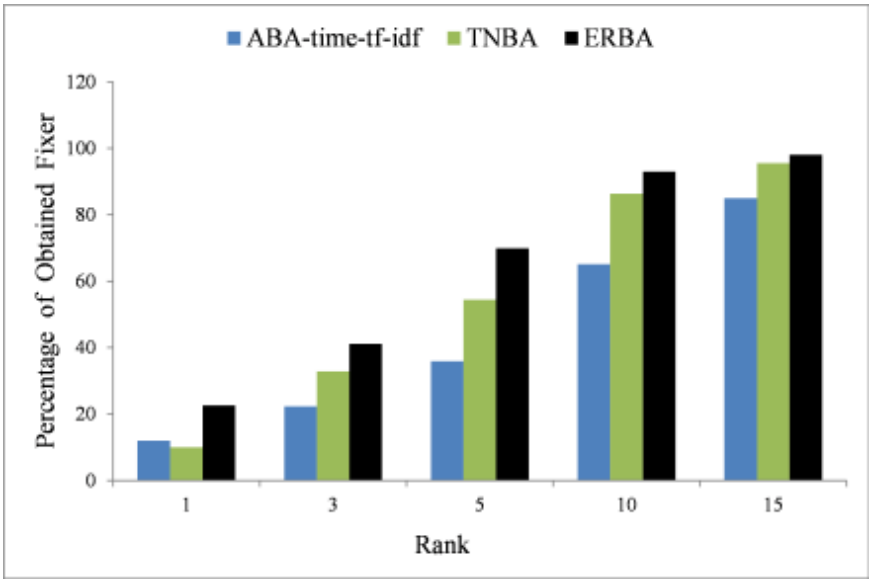

Figure 4.9: Comparison of Top N Ranking on SWT among ABA-time-tf-idf [1], TNBA [2] and ERBA

formance of ERBA is compared with two prominent bug assignment techniques known as ABA-time-tf-idf [1] and TNBA [2]. The performance of both the techniques are compared based on the three mentioned metrics. The comparative analysis of ERBA is explained below.

Figure 4.7, 4.8 and 4.9 show the comparison of Top N Ranking accuracy among ERBA, ABA-time-tf-idf [1] and TNBA [2]. Figure 4.9 shows that for

Table 4.4: Comparison of Average Effectiveness and MRR among ABA-time-tf-idf [1], TNBA [2] and ERBA

Project	**Average Effectiveness**			**MRR (%)**		
	ABA-time-tf-idf	**TNBA**	**ERBA**	**ABA-time-tf-idf**	**TNBA**	**ERBA**
Eclipse JDT	3.24	2.21	2.05	35.18	45.08	48.65
AspectJ	9.2	2.33	2.02	15.90	42.76	43
SWT	3.5	3.45	2.5	33.86	28.97	40.04

each of the rank positions, ERBA achieves higher bar than the existing technique in Eclipse JDT. It retrieves 69.9% fixers within position 5 which is higher than both ABA-time-tf-idf (54.30%) and TNBA (54.47%). A similar trend is also found while Top N ranking in AspectJ and Eclipse JDT (Figures 4.8 and 4.7 respectively). For each of these studied subjects, ERBA outperforms both ABA-time-tf-idf and TNBA while ranking the developers. For these three projects, ERBA on average retrieves about 98.18% of the actual fixers correctly within Top 15 Rank, which is higher than ABA-time-tf-idf (46.33%) and TNBA (97%). These improvement of ranking ensures higher accuracy of ERBA due to combining both recency and expertise information.

Table 4.4 shows the comparison among ABA-time-tf-idf [1], TNBA [2] and ERBA on average effectiveness and MRR. The table shows that ABA-time-tf-idf suggest the first relevant developer on average at rank 3.24, 9.2 and 3.5 for Eclipse JDT, AspectJ and SWT respectively, which are much higher than ERBA, leading to lower effectiveness. The effectiveness value of TNBA is also higher than ERBA. The consideration of expertise information along with the recency information enabled ERBA to achieve higher effectiveness. Besides, improvement in MRR values are gained over both ABA-time-tf-idf [1] and TNBA [2]. For example, an improvement of 13%, 28% and 7% is acheived by ERBA over ABA-time-tf-idf for the three studied projects (Table 4.4). These results depict that consideration of both expertise and recency information leads to higher accuracy of ERBA.

4.6 Summary

In this chapter, an expertise and recency based bug assignment technique has been developed. The technique performs bug assignment using three major steps. The first step extracts the recency information of developers from source code commits and constructs an index for mapping these information. Similarly, the second step constructs another index for mapping the expertise of developers, which are collected using past fixed reports. Finally, the third step formulates a query using the keywords of new bug reports. It then searches the query on the two indexes and combines the query result using tf-idf technique to assign a score to all developers. The top scored developers are suggested as fixers.

The technique is applied on three open source projects for assessing its compatibility. The suggested developers list are evaluated using three metrics and compared with two individual techniques. The experimental results are also discussed briefly in this chapter. Although the technique showed promising results in suggesting accurate developer list, there remains scope for improvement. For example - suggesting team of fixers instead of individuals and associating the newly joined developers in the suggested list. Therefore, in the next chapter a team allocation technique has been devised to ensure bug assignment to both exiting and new developers.

Chapter 5

Team Allocation Considering New Developers

Bug resolution is a collaborative task, and team of developers generally work together to fix a reported bug. Whenever a bug is fixed, the bug solving history is stored in the report by the bug tracking systems. Therefore, automatic fixer team allocation is generally performed by analysing the past fixed bug reports. Due to considering only the past reports, existing approaches fail to accommodate the new developers in the suggested teams. This failure leads to improper task allocation. On the other hand, when a developer joins in a company, he or she may come with a set of expertise. So, a new automatic team allocation approach can be devised using the expertise and recency information to accommodate both existing and new developers. The approach is named as Team Allocation using Exiting and New developers (TAEN). The detailed team allocation procedure of TAEN is explained in this chapter. Experimental results are also presented to evaluate TAEN using two metrics. The evaluation metrics are - recall and workload distribution. A mechanism for balancing the workloads of developers is also developed and presented to analyze its application in team assignment.

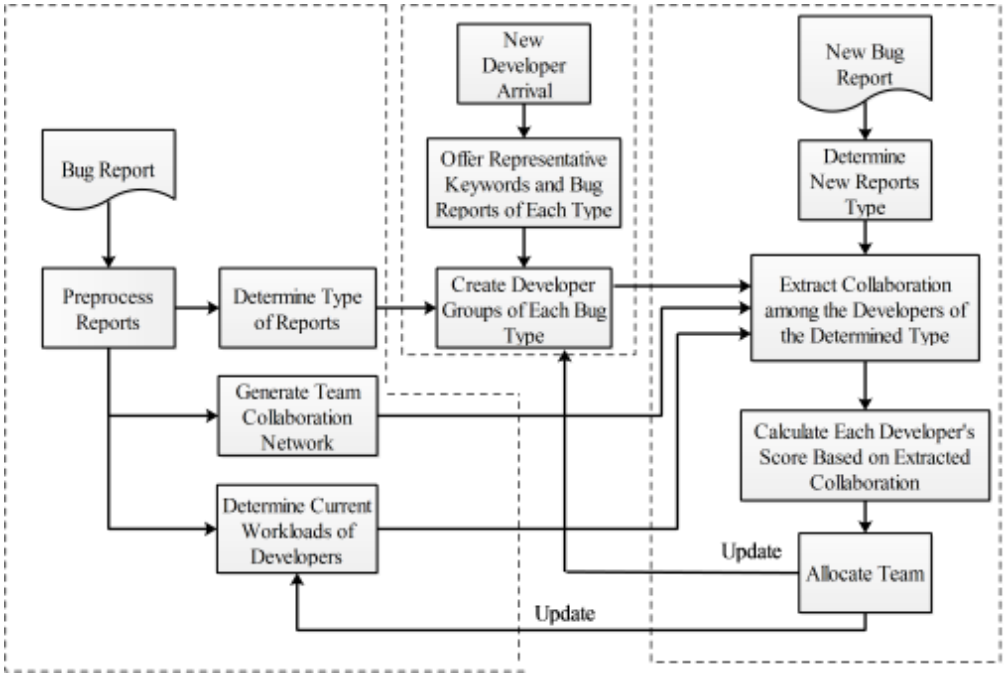

Figure 5.1: Overview of TAEN

5.1 Overview of the Proposed Team Allocation Technique

The overview of TAEN is shown in Figure 5.1. TAEN assigns a fixer team using three main steps. The steps of TAEN are described below -

- **Input Processing:** This step takes previously fixed software bug reports as input as shown in Figure 5.1. The overall functionalities of it are performed using four sub-tasks. Firstly, it pre-processes the bug reports to remove noise and irrelevant terms from the reports. It then represents each report as a document. Secondly, having a collection of documents, it next applies Latent Dirichlet Allocation (LDA), an Information Retrieval (IR) based topic modeling technique to identify possible types of bug reports. Based on these bug types, groups of developers are constructed. The developers who worked on bug reports of same bug type, are grouped under the specific bug type. These produced groups are later used by TAEN as

shown in Figure 5.1. Thirdly, this step constructs a developer collaboration network using four attributes extracted from the preprocessed bug reports. These attributes are - *Developer*, *Bug*, *Component* and *Comment*. The collaboration network is later used for extracting developers' team work while solving previously fixed bug reports. Finally, this step calculates the current workload of each developer. The current workload is determined from the *open* bug reports of the bug repository.

- **Bug Solving Preference Elicitation while New Developers Arrive:** When a new developer arrives, this step offers the developers with the top representative keywords and bug reports of each bug type identified in the last step. The developer can choose the types of bug, they would like to work with. It considers the chosen bug types as the developer's initial bug solving preference. It then adds the new developers with the previously grouped developers of the chosen bug types, as shown in Figure 5.1. This grouping of new developers based on their bug solving preference, allows TAEN to include new developers while final fixer team suggestion.

- **Team Allocation upon Arrival of New Bug Reports:** This step starts processing when new bug reports arrive into the system. It pre-processes the incoming reports, and determines those types in a similar manner mentioned in the *Input Processing* step. It then checks the *severity* property of the newly reported bugs. The *severity* of a bug report can be either of *major*, *blocker*, *critical*, *normal*, *minor*, *trivial* and *enhancement* (Table 5.1). If a report's *severity* is either of *major*, *blocker* and *critical*, this step considers the bug needs to be handled by the existing developers. So, it extracts collaboration among the grouped developers using two factors - *sameBug* and *sameComponent*. These two factors indicate how the developers collaborate on same bugs and same components. Each of these factors

are then identified using three collaboration paths among the developers. A TAEN score is then calculated for each developer using the frequency and recency of those extracted collaborations. The reported bug is then assigned to a fixer team consisting of the top scored N developers. On the other hand, if the *severity* of the reported bug is either of *normal*, *minor*, *trivial* and *enhancement*, both existing and new developers can handle the bug report. This step considers the current workload of the developers, and sorts the developers based on their workload in an ascending order. The N developers who have lower workloads are assigned as fixer team. This inclusion of new developers in bug fixation ensures workload allocation to all developers.

In the following sections, the overall processing of the above three steps is described in detail.

5.2 Input Processing

This step is the input data provider of TAEN. It takes software bug reports in XML format. A bug report contains a number of attributes such as *id*, *status*, *resolution*, *reporter*, *comment*, *component*, *reporting time*, *summary*, *description*, *severity* and *activity history*. These attributes are taken into consideration as they reflect the full solving history of the bugs. For training the LDA model, the bug reports which have *resolved* and *verified* as *status*, and *fixed* as *resolution* property are collected from the bug tracking system. The specification of these properties are enlisted in Table 5.1. Besides, in order to determine the developer's current workloads, the bug reports which have bug *status* either of *new*, *reopened* and *started*, are also input in the system.

The overall processing of this step is divided into four more sub-tasks. These are -

Table 5.1: Bug Report Attributes and Specification [3]

Bug Report Attributes	**Attribute Values**	**Specification**
Severity	Blocker	Blocks other bug report fixing
	Critical	Crashes, loss of data, severe memory leak
	Major	Major loss of function
	Normal	Regular issue, some loss of functionality under specific circumstances
	Minor	Minor loss of function, or other problem where easy workaround is present
	Trivial	Cosmetic problem like misspelled words or misaligned text
	Enhancement	Request for enhancement
Status	New	Currently reported bug
	Assigned	Assigned to any developer for fixing
	Reopened	Reopened as the previous fix did not resolve the bug
	Resolved	A resolution has been performed, and it is awaiting verification by QA
	Verified	QA has verified that the bug resolution is correct
Resolution	Fixed	A fix for this bug is checked into the tree and tested
	Invalid	The problem described is not a bug
	Wontfix	The problem described is a bug which will never be fixed
	Duplicate	The bug is a duplicate of an existing bug
	WorksForMe	More information required to analyze the bug

(a) Report Pre-Processing

(b) Developer Group Creation

(c) Developer Collaboration Network Construction

(d) Developer Workload Determination

In the following, the functionalities of each of these sub-steps are explained briefly.

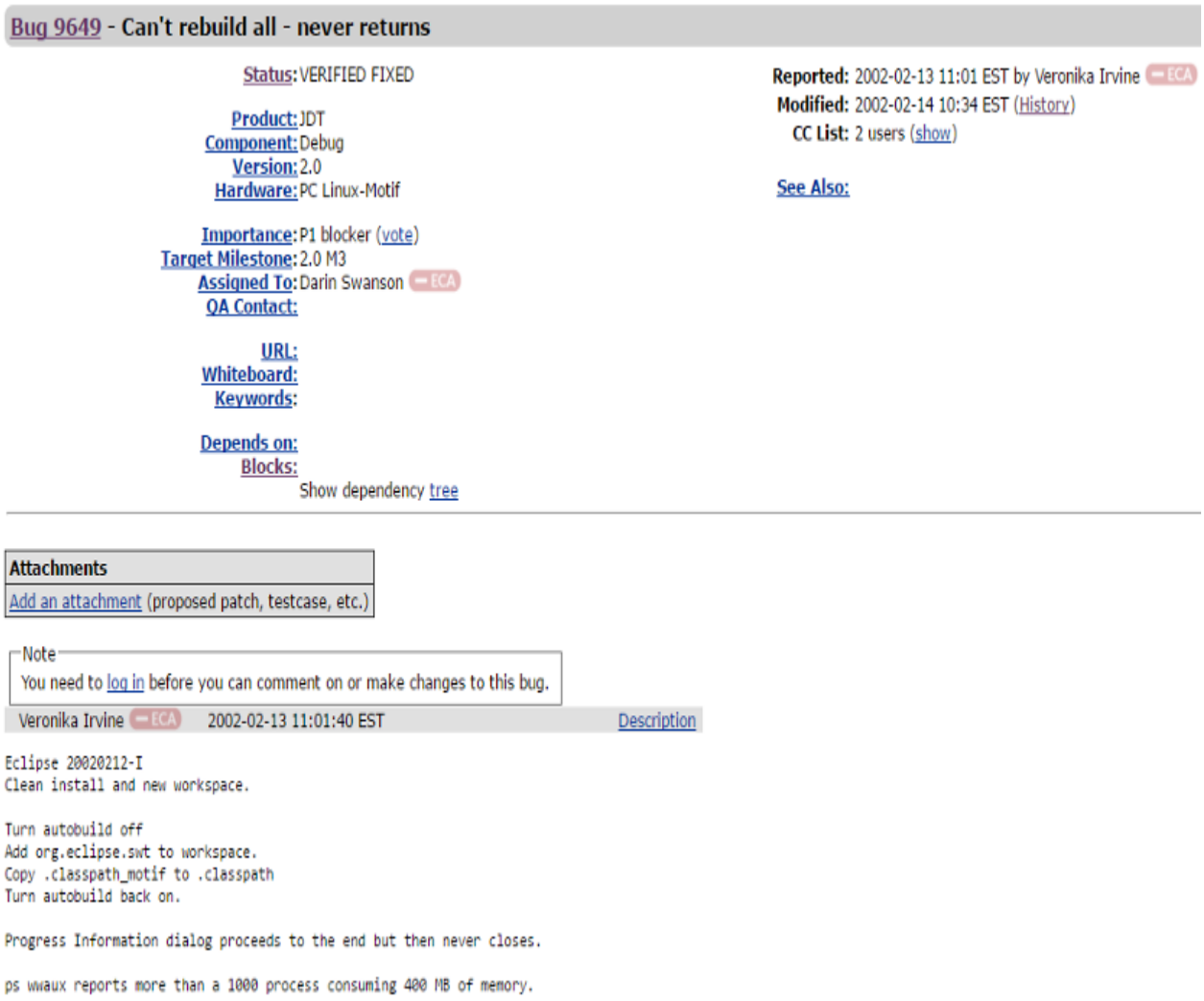

Figure 5.2: Partial Bug Report of Eclipse JDT #9649

5.2.1 Report Pre-Processing

The *summary* and *description* fields of a bug report are given in a narrative format. A snapshot of Eclipse JDT Bug #9649 is given in Figure 5.2. So, these fields may contain a number of noisy and redundant keywords. For better understanding of the field values, several pre-processing are performed which include -

(i) Stop Word Removal

(ii) Stemming

(iii) Keyword Decomposition

(iv) Language Specific Keyword Removal

Each of these pre-processing steps are described in detail in Chapter 2, Subsection 2.4.1.

5.2.2 Developer Group Creation

In object oriented programming, the system is generally developed as combination of different modules. Each module corresponds to a group of functionalities. Therefore, the reported bugs of a specific module may correspond to a specific type. After processing the input bug reports, this step determines the type of each bug report.

For identifying the type of bug reports, LDA modeling is used. Given a list of documents having mixtures of (latent) topics, LDA tends to determine the most relevant topic of the document. So, the *summary* and *description* properties of the bug reports are extracted to represent those as documents. LDA starts processing with those M documents containing a number of word tokens, and having combination of T topics. The main goal of LDA is to find two hidden variables ϕ and θ [66]. Here, ϕ refers the word-topic distribution for each topic. On the other hand, θ refers to the topic distribution over the documents.

LDA first randomly assigns a topic to each word token of each document. It then gradually considers each word token and estimates the probability of the current token's topic, conditioned on the topic assignment to all other word tokens. The probability of a token's topic, $P(z_i = j|z_{-i}, w_i, d_i, .)$ is calculated using Equation 5.1 -

$$P(z_i = j|z_{-i}, w_i, d_i, .) = \frac{C^{WT}_{w_i j} + \beta}{\sum_{w=1}^{W} C^{WT}_{wj} + W\beta} \frac{C^{DT}_{d_i j} + \alpha}{\sum_{t=1}^{T} C^{DT}_{d_i t} + T\alpha} \tag{5.1}$$

the definition of the variables used in this equation is as follows,

- W is total number of tokens in the vocabulary

- T is number of latent topics,

- $z_i = j$ represents the topic j, assigned to current word token ,i

- w_i is a set of word indices,

- d_i is a set of document indices,

- z_{-i} refers topic assignment to all other work tokens,

- . refers to all other known variables such as, w_{-i} is the other word tokens, d_i is the other documents,

- C^{WT} is a $W * T$ dimension matrix, and $C^{WT}_{w_i j}$ indicates the number of times word, w_i is assigned to topic, j without considering w's current instance i,

- C^{DT} is a $D * T$ dimension matrix, and $C^{DT}_{d_i j}$ the number of times topic j is assigned to words of document d_i, without considering the current instance i,

- β is a W dimension vector representing prior weight of word w_i in a topic,

- α is T dimension vector representing prior weight of topic j in a document.

The left part of Equation 5.1 is the probability of word w_i under topic j, whereas the right part is the probability that document d is under topic j. Using these normalized probabilities the ultimate hidden variables are calculated. The probability of word-topic distribution ϕ, is measured using following Equation 5.2 -

$$\phi_{ij} = \frac{C^{WT}_{ij} + \beta}{\sum_{k=1}^{W} C^{WT}_{kj} + W\beta} \tag{5.2}$$

Where, ϕ_{ij} indicates the probability that word token i corresponds to topic j. In a similar manner. the probability of document-topic distribution θ, is calculated using Equation 5.3 -

$$\theta_{dj} = \frac{C^{DT}_{dj} + \alpha}{\sum_{k=1}^{T} C^{DT}_{dk} + T\alpha} \tag{5.3}$$

Here, θ_{dj} represents the posterior probability of document d having topic j. This metric is the variable of interest, to identify the type of each bug report. The

topic for which the bug reports get the highest θ value is taken as the type of the bug report. Thus, using LDA modeling, the type of each bug report is determined.

Once all the bug reports are labeled with one of the T types, the developers who have worked on similar types of bugs are grouped together. Hence, Algorithm 5.1 is proposed for creating developer groups. The *GroupDevelopers* procedure of Algorithm 5.1 takes the processed bug reports as input. This procedure keeps the grouped developers in a complex data structure called *bugTypes* as shown in line 2. The outer map of *bugTypes* links each type to developers who have contributed in that specific type of bugs. The inner map connects each developer name to their contribution frequency on that type of bugs.

A *for* loop is defined at line 4 for iterating on the input bug reports. Each iteration of the loop, first extracts the bug report's type determined by the LDA model. This task is done by calling a method, *GetBugType* as shown in line 5. The method takes the *summary* and *description* of the report, and returns its *type*. The *GroupDevelopers* procedure also extracts and stores the contributor's name of each bug report in a *Set* of strings named *contributors*. Here, the *contributors* refers the reporter and fixers of the bug report as dicussed in Chapter 2, Subsection 2.2.2.

Next, the procedure gets the list of developers mapped against the identified bug *type* (line 7). If no developers are yet mapped against this *type*, a new instance of inner map named *developers* is initialized (line 8-9). All the *contributors* are then populated into the *developers* map which links each developer to their initial contribution frequency (line 10-11). This *developers* map is then put against the identified bug *type* (line 13). On the other hand, if a list of *developers* is already mapped against the identified *type*, another *for* loop is declared for updating the *developers* list (line 15). The loop then checks whether the *developers* list already contains the *contributors* and updates the contribution frequency of

Algorithm 5.1 Developer Group Creation

```
1: procedure GROUPDEVELOPERS(List < BugReport >
       BugReports)
2:     Map < String, Map < String, Integer >> bugTypes
3:     Map < String, Integer > developers, String type
4:     for each b ∈ BugReports do
5:         type ← GETBUGTYPE(b.summary, b.description)
6:         Set < String > contributors ← b.contributors
7:         developers ← bugTypes.get(type)
8:         if developers == null then
9:             developers ← new Map < String, Integer > ()
10:            for each c ∈ contributors do
11:                developers.put(c, 1)
12:            end for
13:            bugTypes.put(type, developers)
14:        else
15:            for each c ∈ contributors do
16:                if devlopers.contains(c) then
17:                    developers.replace(c, developers[c]+1)
18:                else
19:                    developers.put(c, 1)
20:                end if
21:            end for
22:            bugTypes.replace(type, developers)
23:        end if
24:    end for
25: end procedure
```

each contributor, c (line 16-19). Finally, the procedure updates outer map *bugType* with the changed *developers* list (line 22). This procedure continues for all of the input bug reports.

At the end of this stage, each bug type has a group of developers who have previously worked on this type of bugs. These created groups are used later when new bug reports arrive into the system.

5.2.3 Developer Collaboration Network Construction

For solving a reported bug, a number of developers work together as a team. To identify a team of developers, how frequent and recent the developers communi-

Table 5.2: Four Types of Nodes Used in Developer Collaboration Network

Node	**Specification**
D	Developer
B	Bug
C	Component
T	Comment

cate with each other, needs to be determined. While fixing the bugs, developers communicate in various ways such as commenting, verifying, assigning etc. So, this step constructs a developer collaboration network using those communication factors similar to [4]. These communication information are extracted from the input bug reports. The collaboration network is a directed network which is constructed using four types of nodes and eight types of edges. The nodes and edges used in the network are enlisted in following Tables 5.2 and 5.3 respectively.

For example, the first row of table 5.2 shows a node, D which represents the developers involved in the bug fixing process. Similarly B, C and T represent the bugs, bug components and bug comments respectively. Type 1 relationship of Table 5.3 connects a D node to a B node depicting that a developer has *worked on* a bug report. The term *work* refers assignment, report, reassignment, reopening, fixing, verifying or tossing event of a bug. Similarly, Type 4 and Type 5 relations denote that a comment T, *is contained by* a bug B, and a developer D, *has written* the comment T, respectively.

Using those nodes and edges, this sub-step constructs a collaboration network for each of the bug reports. An example network of Eclipse Bug #262605 is shown in Figure 5.3. The Figure shows that the bug is associated with component "Text". It is reported by developer "David Audel". It also shows that the bug report contains a comment having id #1434360 and the comment is written by developer, "Dani Megret".

Table 5.3: Eight Types of Relationships Among Nodes Used in Developer Collaboration Network

Type No.	**Specification**
1	D *works on* B
2	B *is worked on by* D
3	B *contains* T
4	T *is contained by* B
5	D *writes* T
6	T *is written by* D
7	C *contains* B
8	B *is contained by* C

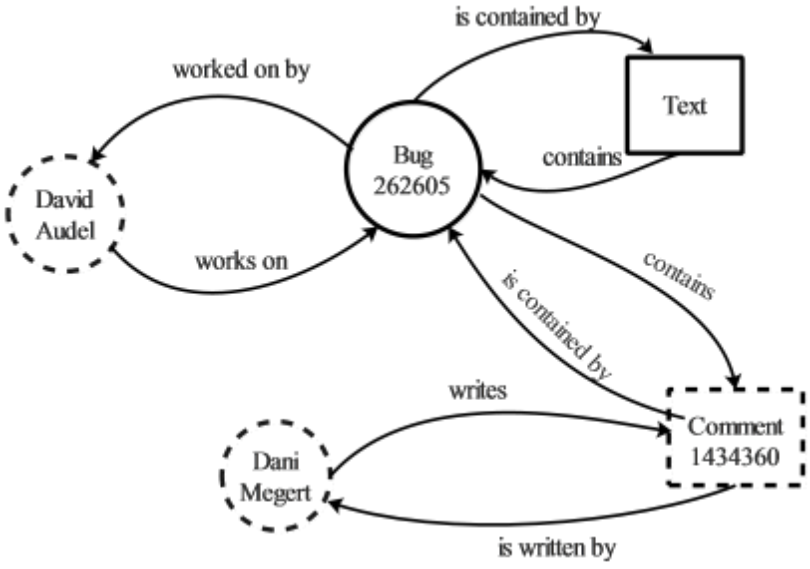

Figure 5.3: A Partial Network of Eclipse Bug #262605

5.2.4 Developer Workload Determination

While assigning bug reports, the existing techniques do not consider the current workload of developers. As a result, the existing developers are always busy with assigned tasks, whereas the new developers may remain idle. As the existing developers would need to fix the bug one after another, it also prolongs bug fixing time. This step collects current workload of developers. The current workload is determined from the *open* bug reports in the bug repository. The bug reports which have *new*, *reopened* or *started* as bug status are considered as *open* bug reports. The workload of each developer is determined by the number of *open* bug reports that he or she is currently assigned with.

5.3 Bug Solving Preference Elicitation while New Developers Arrive

With the passage of time, new developers join the company. These developers do not own any previous fixed bug reports or source commits. In order to assign bug reports to new developers, their initial preference for solving bugs needs to be identified. This step identifies the initial bug solving preference of developers for including them in the team allocation process.

5.3.1 Preference Collection

When a new developer arrives into the system, this step identifies few top representative keywords and bug reports of each bug type learned from the LDA modeling. These list of keywords and bug reports are then offered to the new developers. For example - Table 5.4 shows representative keywords of Type 2 and Type 11 bugs of Eclipse. Finally, the type(s) of bug, the new developer chooses is stored as the initial preference of the new developer. This initial preference

Table 5.4: Few top representative words of bug Type-2 and 11

Type 2	Click	Editor	Select	Display	Dialog	Event
Type 11	Mozilla	Agent	Gecko	Build	User	Windows

updates after the developer fixes any bug(s).

5.3.2 Developer Group Update

After collecting the bug solving preference of the new developers, they need to be attached with the existing developers, to get involved in the bug fixing process. Therefore, an internal buffer group is considered in each of the previously created developer groups in Subsection 5.2.2. The new developers are then added into these buffer groups based on their chosen bug type. The new developers of those buffer groups are later considered in the team allocation process which is discussed in Subsection 5.4.4.

5.4 Team Allocation upon Arrival of New Bug Reports

When a new bug report arrives into the system, this step first processes the new bug report. It then identifies the type of the bug report. The existing and new developers grouped under the identified bug type in Subsection 5.2.2 and 5.3.2 respectively, are extracted. A score is calculated for each existing developer based on how the developers have collaborated for fixing the previous bug reports. Finally, based on the severity of the new report, this step allocates a team of N developers as fixer team. This step performs the overall task using the following four sub steps -

(a) New Bug Report Processing

(b) Developer Collaboration Extraction

(c) Expertise and Recency Combination

(d) Team Allocation

5.4.1 New Bug Report Processing

On arrival of a new bug report B, the type of the report needs to be identified for extracting the developer group to which it can be assigned. The *summary, description* and *severity* properties of the report are extracted and processed. As these property values generally contain irrelevant and noisy terms, pre-processing is done. The processing is done using the techniques mentioned in Subsection 5.2.1.

The new bug report, B is converted into a document having words collected from the *summary* and *description* of the report. The θ for the new bug report is computed from the learned model of Subsection 5.2.2 using the similar Equations 5.1 and 5.3. Finally, the bug type which gets the highest topic distribution value in the θ, is determined as the type of the new bug report. Thus, the developer group associated with this determined bug type is selected as the potential developer group, capable of fixing the recently arrived bug report.

5.4.2 Developer Collaboration Extraction

It is mentioned above that bug resolution is a collaborative task. For recommending a team, the collaboration among the developers needs to be considered. So, this step extracts the collaboration among the developers of the selected group in the previous step.

While solving a bug report, the developers collaborate in various ways. For example - one developer may report the bug, and another developer may fix it. Similarly, a developer may be working on two groups for two different bugs of a same component. Considering those communication patterns, two types

Table 5.5. Six Types of Developer Collaboration

Path Type	**Collaboration on**	**Path**
1	Same Bug	D_1-B-D_2
2	Same Bug	D_1-B-T-D_2
3	Same Bug	D_1-T_1-B-T_2-D_2
4	Same Component	D_1-B_1-C-B_2-D_2
5	Same Component	D_1-B_1-C-B_2-T-D_2
6	Same Component	D_1-T_1-B_1-C-B_2-T_2-D_2

of collaboration factor among the developers are determined. These factors are collaboration over the same bug and same component of the system. Each of these factors is later divided into three paths similar to [4], through which the developers can communicate. These communication paths among developers can be derived from the collaboration network generated in Subsection 5.2.3. The six types of communication paths among the developers are enlisted in Table 5.5.

Type 1, 2 and 3 paths represent how two developers can communicate on the same bug. Similarly, Type 4, 5 and 6 paths identify how two developers can collaborate on the same component. The explanation of these paths are given below -

- **D_1-B-D_2** - a developer (D_1) has worked on a bug (B), which is worked on by another developer (D_2).
- **D_1-B-T-D_2** - a developer (D_1) has worked on a bug (B), which has a comment (T) written by another developer (D_2).
- **D_1-T_1-B-T_2-D_2** - a developer (D_1) has written a comment (T_1) on a bug (B), which contains another comment (T_2) written by another developer (D_2).
- **D_1-B_1-C-B_2-D_2** - a developer (D_1) has worked on a bug (B_1) of a component (C) and another bug (B_2) of the same component (C) is worked on

by another developer (D_2).

- **D_1-B_1-C-B_2-T-D_2** - a developer (D_1) has worked on a bug (B_1) of a component (C), and another bug (B_2) of the same component (C) has a comment (T) made by another developer (D_2).

- **D_1-T_1-B_1-C-B_2-T_2-D_2** - a developer (D_1) has created a comment (T_1) on a bug (B_1) of a component (C), and another bug (B_2) of the same component (C) has a comment (T_2) made by another developer (D_2).

These paths are later considered in the next Subsection for extracting the collaboration history of the developers. Based on these collaborations, TAEN associates a score to each developer for fixer team allocation.

5.4.3 Expertise and Recency Combination

As mentioned before, ignorance of recent activity results in inactive developer recommendation. So, this step adds recency information with the extracted developer's collaboration. The more recent developers work or comment on a bug, the more priority the developers should get. For combining the recent activities, the time when the developers collaborate on the bug, needs to be considered.

Besides, it is already listed in Table 5.1 that a reported bug can have seven types of severity. The developers who solve severe or complex bug reports, should gain more priority than the developers who solve normal ones. Analysing the severity specification of the bug reports, a severity weight is determined for each of the severity values. The severity weights corresponding each severity value are shown in Table 5.6. Thus, the experience of solving bug reports having different severity, also needs to be incorporated while allocating the fixer team. So, Algorithm 5.2 is proposed to compute a score called TAEN score for each developer, by combining the expertise and recency of the collaboration.

Table 5.6: Severity Weights of Bug Reports

Severity Value	**Weight**
Blocker	7
Critical	6
Major	5
Normal	4
Minor	3
Trivial	2
Enhancement	1

The *CalculateScore* procedure of Algorithm 5.2 takes a complex data structure, named *devInfos* as input. This data structure maps the developers to their identified collaboration information of type *DeveloperCollaboration*. Each instance of *DeveloperCollaboration* contains two properties - *sameBugs* and *sameComponents*. The former property contains a list of paths which depicts the associated developer's collaboration on same bugs. Similarly, the later one represents developer's collaboration on same components. The *CalculateScore* procedure represents the partial score calculation process based on the same bugs only. A similar approach is also used for calculating the collaboration score of same components. The procedure starts with defining a data structure called, *devScores* which connects the developers to their calculated TAEN score (line 2). An outer *for* loop is defined for iterating on each developer and an inner loop is defined for iterating on their collaborated paths (line 3-4). Each collaboration path generally connects two developers. Therefore, for each collaborated path the score of the two developers needs to be added or updated (line 5,6). To perform this task, another procedure, *AddScore* is declared (line 10).

This procedure takes an edge and a date as input. It is seen from Table 5.3 that developers directly collaborate by working or commenting on bugs. So, the collaboration edge is sent as parameter for the *AddScore* function. Besides, for adding the recency information of these activities, the collaboration date is

Algorithm 5.2 Expertise and Recency Combination

```
1: procedure calculateScore(Map < String,
       DeveloperCollaboration > devInfos)
2:     Map < String, Double > devScores
3:     for each d ∈ devInfos do
4:         for each path ∈ d.sameBugs do
5:             addScore(path.firstEdge, BugReport.date)
6:             addScore(path.lastEdge, BugReport.date)
7:         end for
8:     end for
9: end procedure
10: procedure addScore(Edge e, Date date)
11:     if e.srcNode == D then
12:         dev ← e.srcNode
13:     else
14:         dev ← e.destNode
15:     end if
16:     if !devScores.keys.contains(dev) then
17:         devScores ← (e.severity/e.fixDuration ∗ (date − e.Date)
18:     else
19:         devScores+ ← (e.severity/e.fixDuration ∗ (date − e.Date)
20:     end if
21: end procedure
```

also sent to this function. It first extracts the developer node from the input edge (line 11-14). It then checks whether the developer has already assigned a TAEN score (line 16). Based on this checking, it adds or updates the score (line 17-19). The score for each collaboration path is initially considered as the severity weight of bug report corresponding the collaboration. Next, this value is divided by two properties - *e.fixDuration* and a date difference (line 17). The former property represents the fixing duration of the report. The higher number of days was spent while fixing a bug report by a developer, the lower score the developer should get. Besides, the date difference represents the difference between the reporting date of the new bug report (*date*) and the collaboration date of the developer (*e.date*). The less large the difference is, the more recent the developer collaborates on bugs, thus the more score the developer gets. Thus, the Lines 17-19 ensures the combination of expertise and recency information of

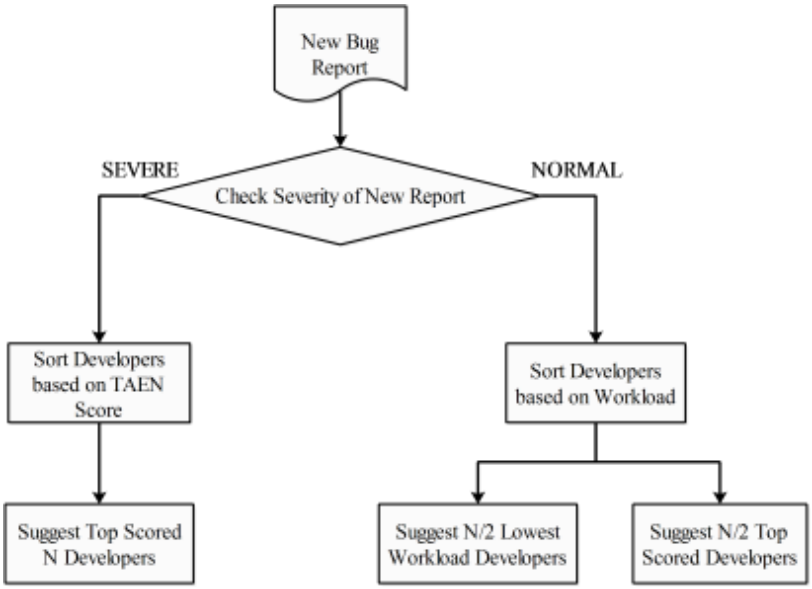

Figure 5.4: Team Allocation Procedure

the developers.

5.4.4 Team Allocation

Finally, using the TAEN score and current workloads determined in the Subsection 5.4.3 and 5.2.4 respectively, this step allocates a bug fixer team. The overall team allocation sub-task is shown in Figure 5.4. It takes the incoming bug report as input. It then extracts and checks the*severity* property of the report. A bug report can have seven types of severity values associated with it as shown in Table 5.1. The first three types *blocker*, *critical* and *major* are generalized as *SEVERE* bug reports. On the other hand, the remaining four types - *normal*, *minor*, *trivial* and *enhancement* are considered as *NORMAL* bug reports.

If the reported bug is identified as *SEVERE*, it is considered that the fixing process is complex, and needs quick resolution as well. So the bug should be handled by existing developers. In this case, TAEN only considers the existing developers. The identified developer group is sorted in descending order based on their TAEN score, calculated in the previous step. The top N developers are allocated as the bug fixing team as shown in Figure 5.4.

If the *severity* of the bug report is identified as *NORMAL*, it is assumed that any developer can fix it. So, TAEN takes all the developers of the group along with the developers stored in the internal buffers created in Subsection 5.3.2. It sorts the developers based on their workload in ascending order. It first selects $\frac{N}{2}$ lower workload developers for allocation of the bug. It then adds top scored $\frac{N}{2}$ developers with the above selected developers to form a team of N fixers. Adding the lower workload developers in the fixer team ensures bug assignment to new developers. Besides, the reason behind allocating existing developers along with new ones is that, assigning tasks only to new developers may prolong bug fixing time. Forming a team with the combination of existing and new developers enables the new developers to learn from the existing developers. When a bug report would be fixed, the new developers would be removed from the internal buffers, and added to the group of existing developers. The developer collaboration network is also updated accordingly upon fixation of a bug report. These incremental updates ensure profile building of developers.

5.5 Result Analysis

In order to measure the applicability of TAEN, experimental analysis is performed on three open source projects. These are - Eclipse Java Development Tool (JDT) [63], Netbeans [67] and AspectJ [64]. All these projects are taken into consideration due to two reasons. Firstly, these projects have been used by other existing techniques ([1, 4]). Secondly, the bug repository of these projects are available in open source. The experiments are performed to evaluate whether the new developers are allocated during bug assignment. Besides, the inclusion of actual fixers in the allocated teams are also evaluated. The team allocation of TAEN is evaluated using two metrics - Recall and Workload distribution. The results are compared with an existing team based bug assignment technique,

known as KSAP [4]. The overall experimental procedure is briefly discussed in the following Subsections.

5.5.1 Data Collection

The details about the collected datasets are mentioned below-

- **Eclipse JDT:** JDT project is a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse platform. The details about JDT project is discussed in Chapter 4, Subsection 4.5.1. The JDT project maintains its bug repository using Bugzilla [16]. Thousands of bug reports are maintained and available in the Eclipse Bugzilla repository [68]. Here, total 2676 bug reports from year 2009 to 2016, are collected for experimental analysis as shown in Table 5.7. Among these reports, 2000 bug reports have *resolved* or *verified* as bug *status*, which are used for the training and testing purposes. On the other hand, for determining the current workload of developers, 676 bug reports which have *new* or *assigned* as bug *status* are collected.

- **Netbeans:** Netbeans is an open source project dedicated to support software development that address the needs of developers. It is basically a widely used software development platform written in Java. The two base products of Netbeans are - Netbeans IDE and Netbenas Platform. The NetBeans IDE was developed mainly for Java, but also supports C, C++ and dynamic programming languages. Netbeans platform runs the applications developed using the IDE. The bug reports of Netbeans are also maintained using Bugzilla [69]. Total 2916 bug reports of Netbeans between years 2009 and 2016 are collected for evaluation. 2267 *resolved* or *verified* bug reports are extracted for learning purpose of TAEN as shown in Table 5.7. Besides, 649 *new* or *assigned* bug reports are also used for

Table 5.7: Details of Experimental Dataset

Project	Year Duration	No. of Bug Reports		Total No. of Bug Reports
		Resolved/Verified	New/Assigned	
Eclipse JDT	2009 - 2016	2000	676	2676
Netbeans	2009 - 2016	2267	649	2916
AspectJ	2004 - 2016	1522	560	2082

Figure 5.5: A Snapshot of Eclipse JDT Bug #509075

measuring the current workload of developers.

- **AspectJ:** AspectJ is also selected as a study object for evaluation which is already described in Chapter 4, Subsection 4.5.1. The bug reports of AspectJ are managed under the tools option of Eclipse bug reports [70]. For experimental analysis, 2082 bug reports from year 2004 to 2016 are selected. Here, 1522 *resolved* or *verified* bug reports are assembled for evaluation purposes. The assigned workload of each developer is calculated by analyzing 560 *new* or *assigned* bug reports.

Bugzilla supports exportation of bug reports in XML format. So, all those bug reports are automatically exported in XML format. A set of *resolved* or *verified* bug reports are randomly selected for evaluating the team allocation of TAEN. The allocated team members for each test bug report, are checked against the actual fixers set of the report. The actual fixers set of a report is obtained

Who	When	What	Removed	Added
markus_keller	2016-12-13 05:34:14 EST	Assignee	jdt-core-inbox	sxenos
		Target Milestone	---	4.7 M5
sxenos	2016-12-13 09:15:31 EST	Priority	P3	P1
genie	2016-12-13 10:03:51 EST	See Also		https://git.eclipse.org/r/87053
genie	2016-12-13 10:09:25 EST	See Also		https://git.eclipse.org/c/jdt/eclipse.jdt.core.git/commit/?id=
sxenos	2016-12-13 10:10:23 EST	Status	NEW	RESOLVED
		Resolution	---	FIXED
jarthana	2017-01-24 10:53:48 EST	Status	RESOLVED	VERIFIED
		CC		jarthana

Figure 5.6: The History of Eclipse JDT Bug #509075

from the activity history of the report published and verified by Bugzilla. For example, Figure 5.5 shows snapshot of a *verified* and *fixed* bug report of JDT #509075[1]. The corresponding fixing history[2] of the report is shown in Figure 5.6. It is clearly seen from Figure 5.6 that developer "markus_keller" assigned the report to "sxenos". Here, "sxenos" first changed the priority of the report. Meanwhile, developer "genie" provided "sxenos" with some urls for the bug resolution. Then, "sxenos" worked on the report and changed its *status* from *new* to *resolved*, and set the *resolution* as *fixed*. Finally, developer "jarthana" verified the bug resolution as a correct one. The bug report also contained comments[2] from these developers. Therefore, these developers are taken as the actual fixers of the bug report #509075.

The above mentioned collected data of the three projects are available in the git repository of TAEN [71].

5.5.2 Evaluation Metrics

The performance of TAEN is calculated using two metrics - Recall@N and Workload distribution. Recall is the most commonly used metric for assessing the applicability of an allocation or suggestion system. A higher value of this metric indicates higher applicability of TAEN. On the other hand, none of the existing

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=509075
[2]https://bugs.eclipse.org/bugs/show_activity.cgi?id=509075

techniques assigned tasks to new developers. Therefore, the number of workloads assigned to each developer by TAEN is measured to represent TAENs compatibility in preference based bug assignment to new developers. The details of these metrics are discussed below.

- **Recall@N:** Recall@N refers the fraction of relevant developers placed in the N sized allocated team. Here, the relevant developers are the actual fixers of the bug report. The higher the number of actual developers included in the top N places is, the more correct the allocation is. The recall is calculated using Equation (5.4) similar to [4] -

$$Recall@N = \frac{|\{dev1, dev2, ..., devN\} \cap \{GroundTruth\}|}{|GroundTruth|} \tag{5.4}$$

  Here, $\{dev1, dev2, ..., devN\}$ is the set of N developers allocated for resolving each test bug report. $\{GroundTruth\}$ refers to the set of actual developers who contributed in fixing the bug report in real practice. The $\{GroundTruth\}$ set of each test bug report is collected from the Bugzilla as mentioned in the previous Subsection.

- **Workload or Task Allocation:** Workload allocation refers to the number of tasks allocated to each developer. As none of the previous techniques assigned tasks to new developers, the number of tasks allocated to new developers by TAEN is calculated. If the assigned workload of a developer is 0, it represents improper task allocation. An increase in the number of tasks assigned to new developers ensures inclusion of new developers in the allocated teams by TAEN.

5.5.3 Research Question and Evaluation

In this Subsection, TAEN is evaluated by answering the second research question of Chapter 1, Section 1.3. The answer of this research question is obtained by

achieving two goals - **Goal1** and **Goal2**. **Goal1** states how many actual fixers are successfully included in the allocated teams by TAEN. Moreover, **Goal2** demonstrates TAENs capability of allocating bug reports to new developers as well. The detailed description of these goals along with evaluation results is explained below.

Goal1: How many actual fixers are successfully included in the allocated teams?

For achieving this goal, three open source projects are used as mentioned in Table 5.7. For each test bug report, a team of N = 3, 5, 7 and 10 developers is allocated for bug assignment. The more number of actual fixers are included in the allocated teams, the more correct the allocation is considered. This allocation is measured using Recall@N as mentioned in Subsection 5.5.2. Along with this, the recall rate of TAEN is compared with an existing team assignment technique known as KSAP [4]. The recall rate of TAEN on the three open source projects are demonstrated below.

Among the 2000 *fixed* bug reports of Eclipse JDT as shown in Table 5.7, 1250 bug reports were used for training of TAEN. These bug reports were divided into 17 distinct topics which are also used in [13]. However, Various techniques are available in the literature for identifying the natural number of topics when applying LDA [13]. Again, the number of topics for each project is different. So it can also be set by the bug triager based on the project repository. Randomly 750 bug reports were used for testing purposes. These bug reports contain 600 unique developers. Besides, these bug reports contain on average 3.58 unique developer collaborations. All the 750 bug reports are tested for suggesting a team of N = 3, 5, 7 and 10 developers respectively. The average recall rates of Eclipse JDT when allocating different size of teams are shown in Figure 5.7. It shows that on average 40.14% of the relevant developers are placed in the allocated teams when

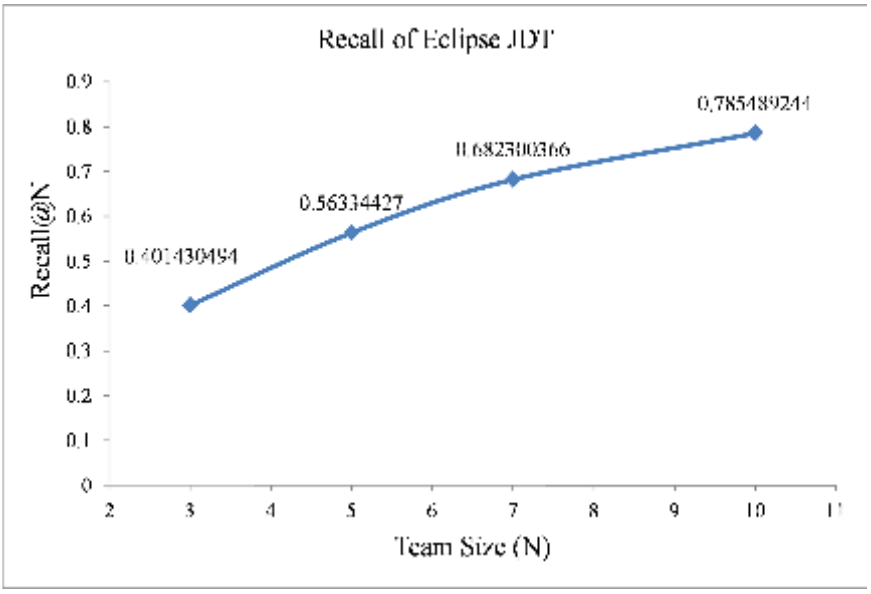

Figure 5.7: The Recall@N rate of TAEN in allocating different number of developers on Eclipse JDT

suggesting a team of 3 developers. This recall rate increases linearly to 56.33%, 68.23% and 78.54% with the increase in team size to 5, 7 and 10 respectively. These higher values of recall rate indicate that TAEN is capable of assigning new bug reports to the actual expert and recent developers. It also depicts that TAEN suggests fixer teams with combination of both new and existing developers, which would help faster bug resolution as well as profile construction of new developers.

In a similar manner, 767 *fixed* bug reports are selected randomly for analyzing TAENs performance on Netbeans. These bug reports were divided into 30 distinct topics. The reason behind selecting 30 is that, the Netbeans source repository contains 30 major modules [72]. As object oriented programming generally develops a system as a collection of modules, each module is dedicated to particular functionalities. So the bug reports corresponding to a system module is considered of same type. The collected dataset of Netbeans includes information about 600 unique developers. An average of 5.02 unique developers take part in comments and fixation of the reports. A team of 3, 5, 7 and 10 developers are allocated for each of the 767 bug reports. The recall of the allocated teams

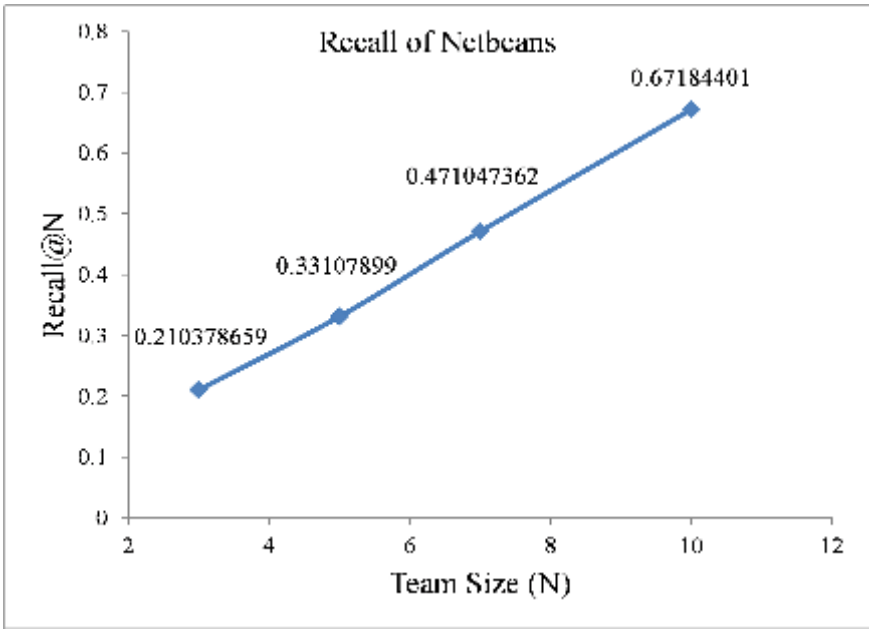

Figure 5.8: The Recall@N rate of TAEN in allocating different number of developers on Netbeans

are plotted in Figure 5.8. A similar linear growth in the recall rate is also found for the test reports of Netbeans. 67.18% of the relevant developers are suggested when $N = 10$, when assigning the test bug reports. This higher value of this metric shows correct team suggestion by TAEN. Figure 5.8 also shows that, for the teams of size 3, 5 and 7, the recall rate of 21.03%, 33.10% and 47.10% are gained respectively on Netbeans.

For AspectJ, 522 *fixed* bug reports are randomly selected as test reports for applying on TAEN. The reports are divided into 17 distinct topics [73]. These reports contain information regarding 378 unique developers. It is also seen that on average 3 developers collaborate by commenting and working for resolution of a bug report. The recall rates while allocating a team of 3, 5, 7 and 10 fixers, for each of the test reports are shown in Figure 5.9. Here, 59.14%, 59.70%, 60.47% and 61.27% recall are achieved when suggesting teams of 3, 5, 7 and 10 developers respectively. The linear growth of the line in Figure 5.9 is more stable, which indicates that TAEN places most of the relevant developers in the top 3 positions. Placing the relevant developers at the top of the suggestion indicates

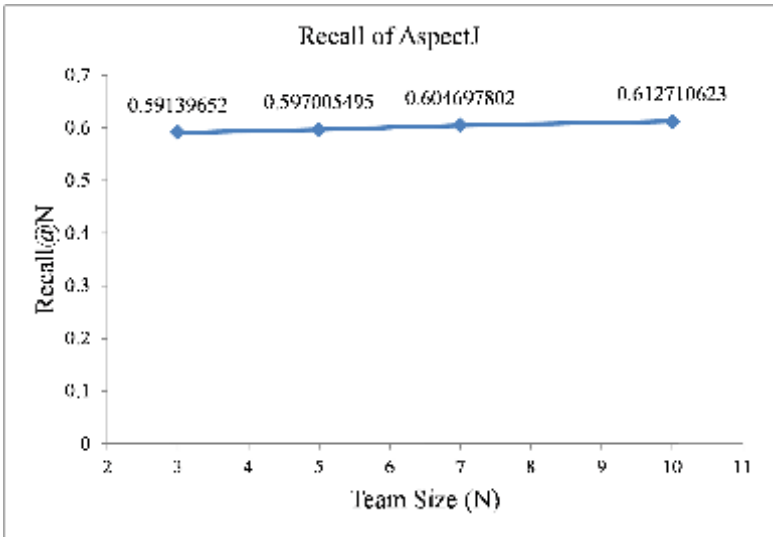

Figure 5.9: The Recall@N rate of TAEN in allocating different number of developers on AspectJ

higher effectiveness of TAEN.

The average recall rate achieved by TAEN is compared with an existing team suggestion technique named KSAP [4]. Table 5.8 illustrates the average recall when allocating a team of 10 developers achieved by TAEN and KSAP. The table clearly shows that for each of the three projects, TAEN achieved a higher average recall than KSAP. For example, for Eclipse JDT, TAEN successfully retrieved 78.55% of the actual fixers. On the other hand, KSAP retrieved 65.88% actual fixers which is lower than TAEN. Along with this, 67.18% and 61.27% recall is achieved for Netbeans and AspectJ, which also indicates effective team allocation by TAEN. Consideration of both recent and previous activities enabled TAEN to improve the recall than KSAP. These results show that TAEN is capable of allocating relevant fixer teams for newly arrived bug reports.

Goal2: Are the new developers included while allocating teams for newly arrived bug reports?

While assigning bug reports, TAEN considers the new developers along with the existing ones. Existing approaches assign bug reports to the most experienced developers. However, TAEN assigns bug reports both to the existing and new

Table 5.8: Comparison of Recall Rate between TAEN and KSAP [4]

	Approach	
Project	**KSAP [4]**	**TAEN**
Eclipse JDT	65.88	78.55
Netbeans	55.51	67.18
AspectJ	50.17	61.27

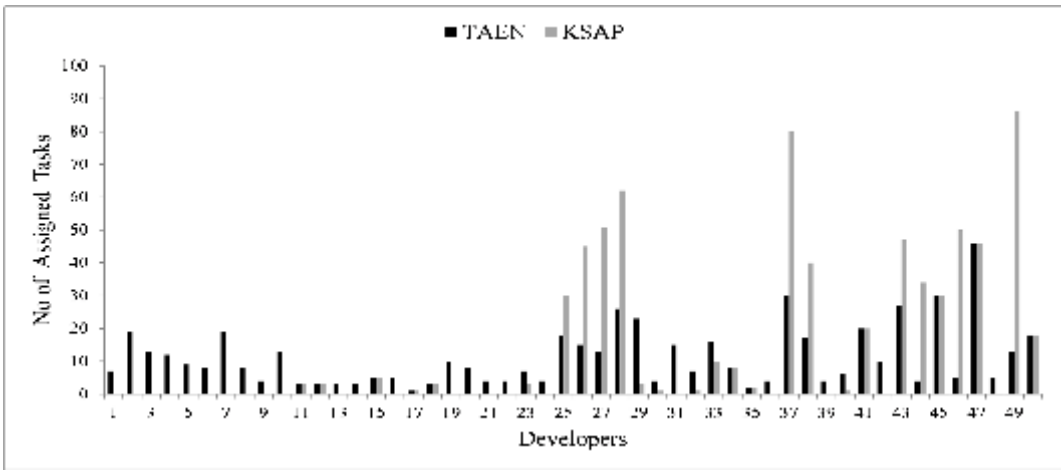

Figure 5.10: Comparison of Task Allocation between TAEN and KSAP [4] on Eclipse JDT

developers based on the *severity* of the reports. So, after assigning all the test reports for each of the three studied projects, the number of bug reports assigned to the developers are analyzed. The developers who do not have any past history, that is, they are not grouped under any of the bug types are considered as new developers in the experimentation. The bug solving preference of these new developers is determined by the type of bugs they are currently assigned to. Based on this preference, these developers are initially grouped with one of the bug types. The assigned workload of each developer by TAEN is compared with the existing technique, KSAP [4]. The workload distribution of the developers for each of the three projects are demonstrated in the following.

After allocating teams for all the 750 bug reports of Eclipse JDT, the current workload of each developer is counted. These 750 test reports are applied on both TAEN and KSAP. Figure 5.10 shows the current workloads of 50 unique

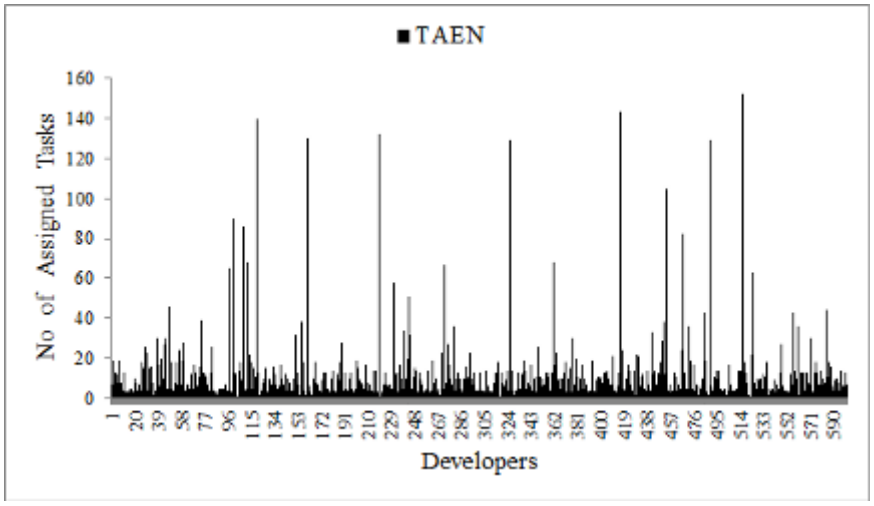

Figure 5.11: Workload or Task Allocation among 600 developers by TAEN on Eclipse JDT

developers assigned by TAEN and KSAP. The bar chart of Figure 5.10 clearly shows that KSAP assigns no bugs to the new developers plotted at the left of the graph, that is the new developers are unused. However, TAEN successfully assigns bug reports to the new developers as well based on their preference. Besides, as KSAP ignores the new developers, the experienced developers get overloaded with queue of bugs. For example, both developer No. 37 and 49 of Figure 5.10 are overloaded with above 80 tasks or bug reports. On the other hand, TAEN assigns 30 and 45 bug reports to those developers respectively, which is lower than KSAP. It is so because TAEN allocates task to the new developers as well. The black bars of Figure 5.10 indicates that all the developers are assigned tasks based on the severity of the test reports. These results indicate proper task allocation to both new and existing developers.

The assigned workloads to all the 600 unique developers of Eclipse JDT is plotted in Figure 5.11. It illustrates that almost all the developers are considered while assigning tasks. Some of the developers have higher workloads. The reason behind is that the severe bug reports need assistance of expert developers. Without the assistance of experienced developers, the resolution of these complex reports would prolong the bug fixing time. So, TAEN assigns severe reports

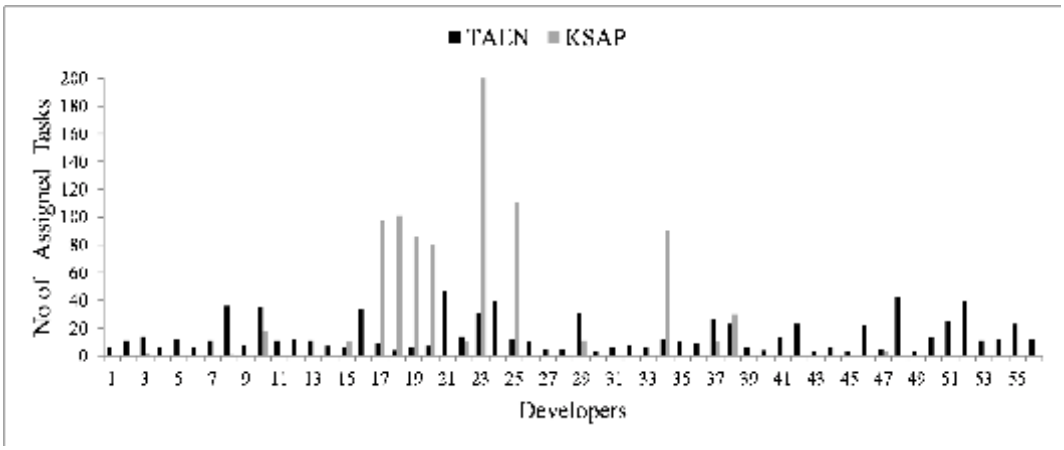

Figure 5.12: Comparison of Task Allocation between TAEN and KSAP [4] on Netbeans

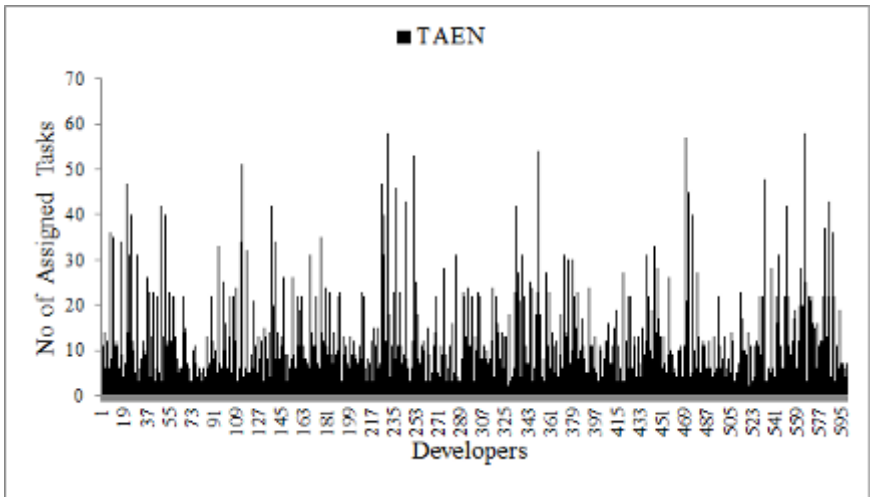

Figure 5.13: Workload or Task Allocation among 600 developers by TAEN on Netbeans

to the experienced developers. In other cases, TAEN assigns tasks to both the existing and new developers as shown in Figure 5.11.

In case of Netbeans, after applying the 767 test bug reports, the number of tasks assigned to the developers is measured similarly. The collected reports involve 600 unique developer contributions. The assigned tasks of 50 random unique developers are illustrated using the bar chart shown in Figure 5.12. This figure also indicates that TAEN assigns tasks to new developers whenever *NORMAL* bug report arrives in the system. On the other hand, KSAP assigns no task to the new developers shown at the left and right ends of the graph. The allocated tasks to all the 600 unique developers of Netbeans is plotted in Figure 5.13. A more stable task allocation is seen here, where all the developers are associated with bug reports. These results show proper task allocation by TAEN, which would allow incremental profile building of the newly joined developers.

Similarly, for AspectJ 522 *fixed* bug reports are applied on both TAEN and KSAP. The tasks allocated to the 378 unique developers of AspectJ are obtained. Figure 5.14 shows the comparison of task assignment between TAEN and KSAP, while assigning tasks to 50 developers of AspectJ. The right end of the bar chart shows that the new developers are getting tasks incrementally by TAEN, whereas KSAP assigns tasks only to the existing developers. For example - developer no. 17 gets 150 tasks by KSAP, whereas developers from no 22 to 27 remain unallocated. However, TAEN assigns tasks to all of these developers.

The number of tasks allocated to the 378 unique developers of AspectJ are plotted in Figure 5.15. This figure shows a pattern similar to Eclipse JDT. All the developers are allocated with some tasks and some of the developers are assigned higher number of tasks. The reason behind it is, among the 522 test reports, 280 bug reports were identified as *SEVERE*. These reports are assigned to expert developers, as they require complex fixing. For rest of the cases, TAEN assigns task to both existing and new developers. Thus, the smaller frequent bars

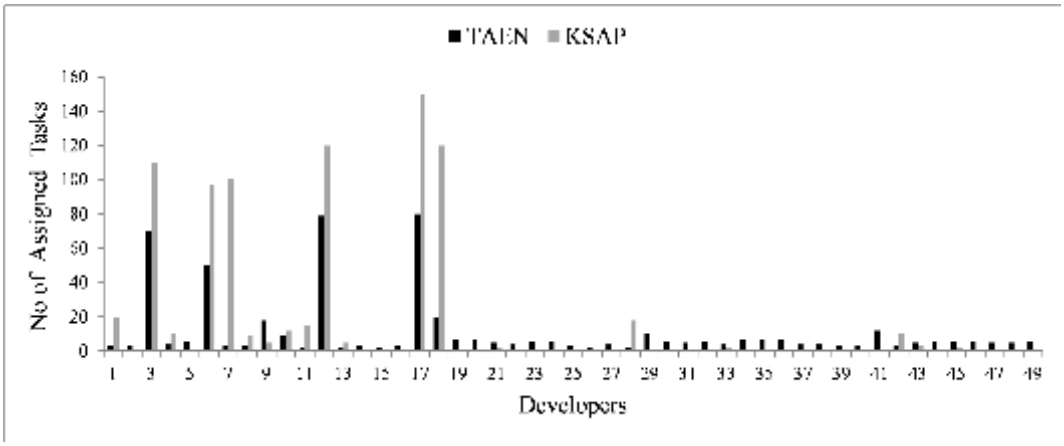

Figure 5.14: Comparison of Task Allocation between TAEN and KSAP [4] on AspectJ

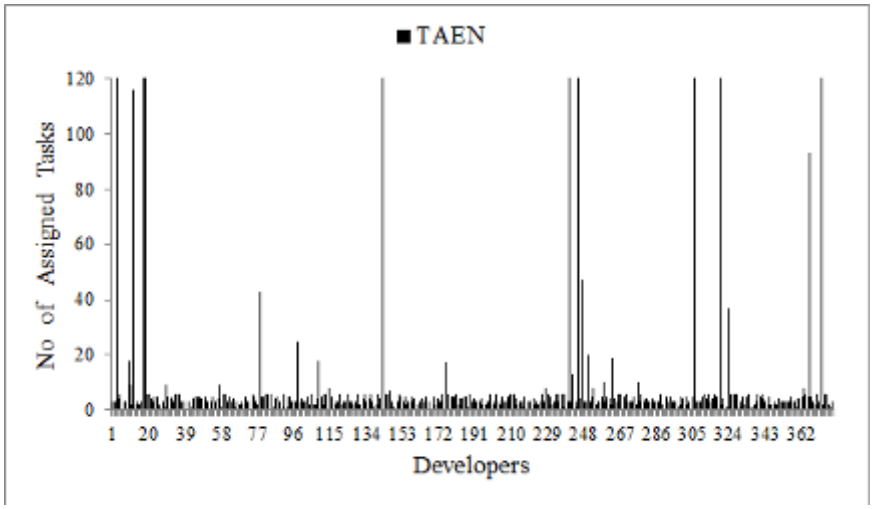

Figure 5.15: Workload or Task Allocation among 378 developers by TAEN on AspectJ

show consistent task allocation to all the developers.

5.6 Balanced Task Allocation

The existing bug assignment techniques focus to allocate tasks to developers who can resolve it best within the time schedule. During the process of finding experienced developers, these techniques totally ignore the new developers. This is because, the available information sources, using which the existing techniques measure the experience and recency of developers, do not contain any record regarding the new developers. To mitigate this limitation of task assignment to new developers, a team based bug assignment technique named as TAEN is proposed in this chapter. The compatibility of the technique is evaluated using two metrics - recall and workload allocation. The experimental results of Section 5.5 show that TAEN is effective in terms of allocating tasks to actual developers as well as to all developers, which supports the main research question of this thesis. However, while analyzing the task assignment to developers it is seen that, the most experienced developers may get overloaded if *SEVERE* bug reports are reported frequently. Therefore, imbalanced task allocation may occur. To overcome this situation, a current workload checking mechanism is added in the *Team Allocation* step of TAEN to analyze its effect on bug assignment. In this section, this variant of the *Team Allocation* procedure, and its application on the three open source projects are demonstrated.

The workload allocation to all the developers for the three studied projects are shown in Figure 5.11, 5.13 and 5.15. These figures indicate that for each project, few developers get a higher number of tasks than the other developers. For example, Figure 5.15 shows that few developers get above 100 tasks to be resolved, whereas most of the developers have around 5 bug reports to be done. From the perspective of bug assignment, this case is an expected scenario, as

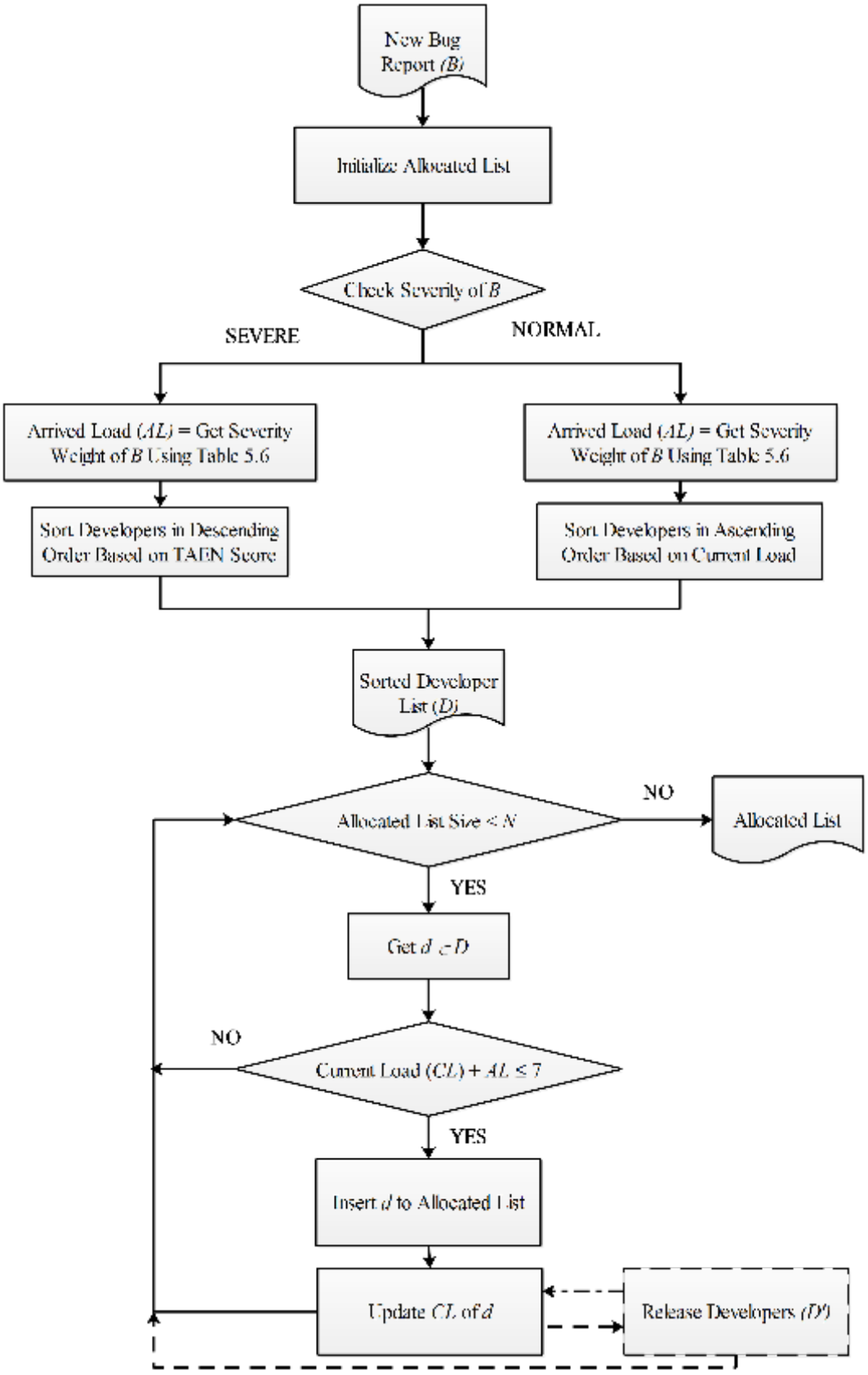

Figure 5.16: Transformed Team Allocation Process by Balancing Workload

complex tasks need assistance of expert developers. Besides, for faster and correct bug resolution, allocating tasks to appropriate fixers is essential. However, from the perspective of developer utilization, this scenario may lead to over-specialization or over-utilization of developers. Considering these factors, a load checking mechanism is added to the *Team Allocation* procedure of TAEN.

After adding the load checking mechanism, the transformed *Team Allocation* procedure is shown in Figure 5.16. Table 5.1 shows that, type *blocker* is the most severe bug. If a developer is currently assigned to a most severe bug, the developer cannot take additional tasks. Thus, the maximum workload of a developer is assumed to be 7 using Table 5.6. The procedure of Figure 5.16 takes the new bug report B, as input. It then initializes the final allocation list and checks the severity of B. The bug report can be identified as either of *SEVERE* and *NORMAL* as mentioned in Subsection 5.4.4. Next, the severity weight of B is determined and stored as Arrived Load *(AL)* using Table 5.6. If the bug is identified as *SEVERE*, it is assumed that the bug needs to be handled by experienced and recent developers. So, the developers are sorted in descending order of their TAEN scores as shown in left side of Figure 5.16. Otherwise, the bug report is assumed to be handled by all developers, and so the developers are sorted in ascending order of their Current Load *(CL)*. As a result, a sorted developers list is found and input in the team creation process.

For creating the fixer team, it is first checked whether the initialized allocation list is populated with N fixers. If not, this procedure gets the top developer d, from sorted developers list and checks whether the arrived load AL, would over flow the maximum limit of a developer. If not, the developer is inserted in the final allocation list and the current load of the developer is updated. Otherwise, this process continues with the next top developer until a N sized fixer team is generated for bug assignment.

A started task or bug report is naturally to be resolved or fixed at any partic-

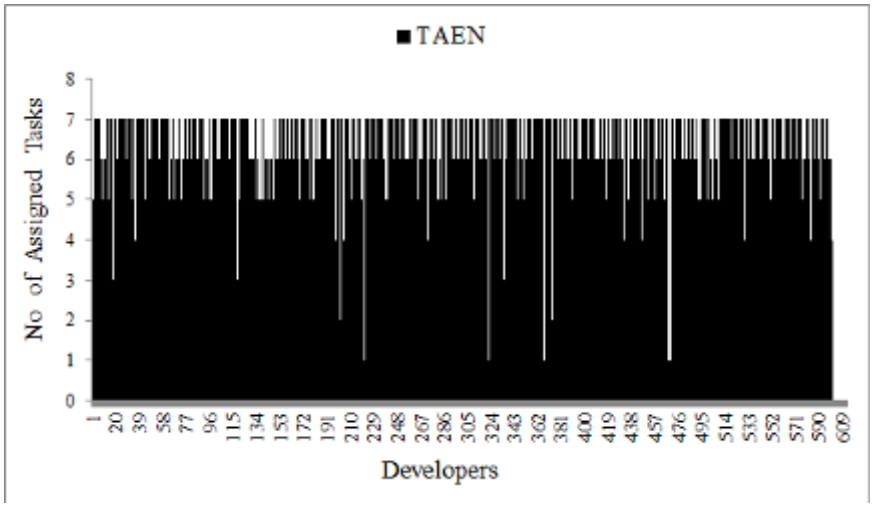

Figure 5.17: Balanced Workload or Task Allocation on Eclipse JDT

ular time. Considering this factor, a step for releasing developers is also modeled in the *Team Allocation* procedure as shown in right bottom part of Figure 5.16. Whenever a bug is resolved, the corresponding developer team will be released and their current load will be updated. At the end, this procedure allocates a team of N developers.

The same set of test reports as mentioned in Subsection 5.5.3, are applied on the three projects using the diverted *Team Allocation* procedure. After assigning each test reports, a random group of N developers is released, as the actual resolution time of a new bug report is unknown. The number of tasks assigned to all developers are measured and plotted similarly. Figure 5.17, 5.18 and 5.19 show the number of assigned loads to each developer. All these figures show balanced task assignment to all the developers. It is evident from the figures that the current load of a developer never reaches above 7. Besides, the number of tasks allocated to each developer remains consistent around 0 to 7.

For understanding the balanced task assignment by the transformed *Team Allocation* procedure, the utilization rate of the tasks allocated to each developer

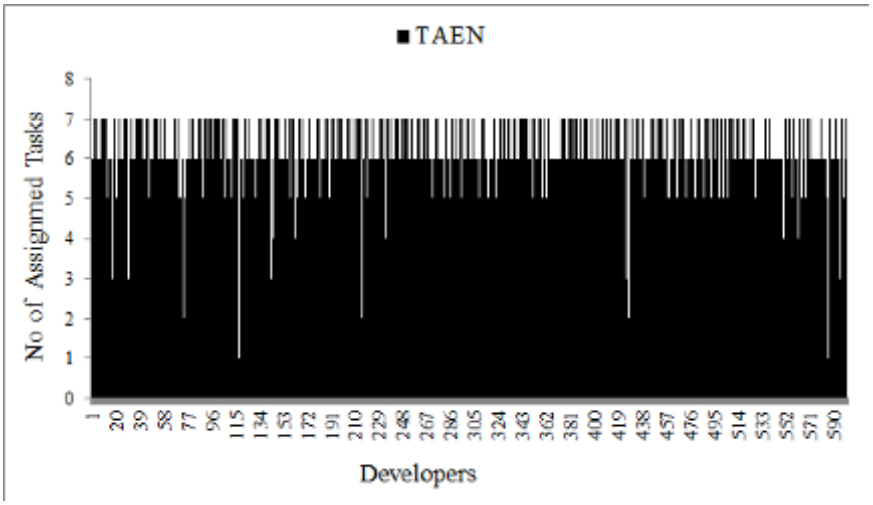

Figure 5.18: Balanced Workload or Task Allocation on Netbeans

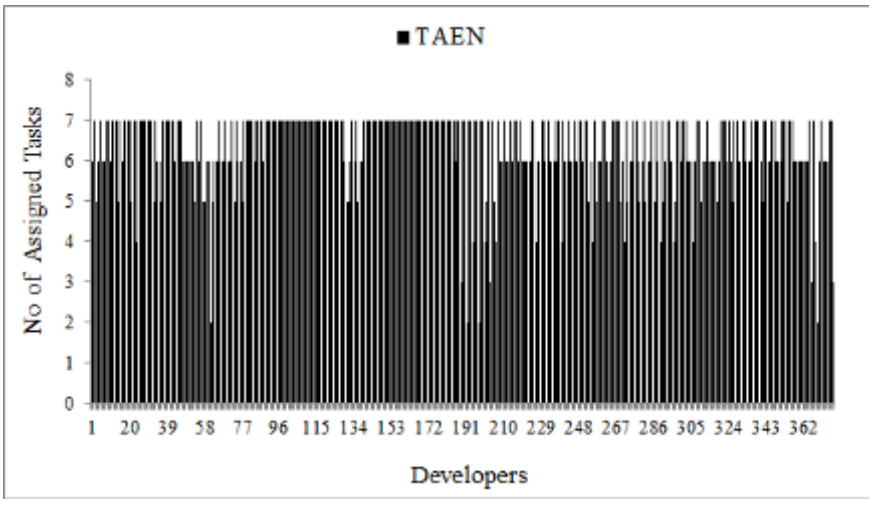

Figure 5.19: Balanced Workload or Task Allocation on AspectJ

Table 5.9: Utilization Rate of Developer Workload Assignment

Project	**Average Utilization (%)**
Eclipse JDT	85.51
Netbeans	89.24
AspectJ	98.11

is measured. The utilization rate is measured using the following Equation 5.5-

$$UtilzationRate(d) = \frac{assignedLoad(d)}{maximumLoad} \tag{5.5}$$

Here, $assignedLoad(d)$ refers to the number of tasks allocated to developer d and $maximumLoad$ refers the highest number of loads a developer can be assigned. A higher value of this metric represents higher utilization of the developers. For each project, the utilization rate of all the developers are computed and the average of these rates is enlisted in Table 5.9. For example - the average utilization rate of AspectJ is 98.11% which indicates that the developers of AspectJ are on average allocated with 98.11% tasks.

The above mentioned findings indicate that the transformed *Team Allocation* procedure can balance the workloads among the developers. However, if the expert developers are fully loaded with tasks, this procedure assigns tasks to team of non-expert developers. Thus, it fails to assign task to appropriate developers, which degrades the recall of bug assignment techniques. So, there exists a trade-off between balanced task assignment and correct bug assignment. The main goal of bug assignment technique is to assign tasks to appropriate developer(s) for faster and correct resolution. Combining the goal of balancing tasks with correct bug assignment opens another research issue where the concept of the transformed *Team Allocation* procedure can be used.

5.7 Summary

In this chapter, a team allocation technique for assigning bug reports to both existing and new developers is proposed. The technique performs the overall team allocation using three steps. The first step takes bug reports as input and applies Latent Dirichlet Allocation modeling to identify the type of each bug report. Next, the developers who worked on similar bug types are grouped together. A developer collaboration network is constructed using four types of nodes and eight types of edges extracted from the input bug reports. This step also measures the current workload of all the developers. Next, on arrival of new developers, the second step determines their bug solving preferences by representing top representative keywords and bug reports to them. Lastly, when new bug reports arrive, the last step determines the type of the report to identify potential developers of this type. A collaboration score is measured for each developer by analyzing six types of communication through the network, among the potential developers. Finally, a team of N developers is allocated checking the severity of the bug report.

In order to assess the applicability of TAEN, its application on three open source projects is also discussed in this chapter. Two research questions are explained to demonstrate how many actual fixers are allocated and whether the new developers are considered for team allocation. Experimental results are also presented. A workload balancing mechanism is also applied on the team allocation procedure, to analyse its effectiveness while assigning bugs. The next chapter concludes the whole thesis with possible future remarks.

Chapter 6

Conclusion

Automatic team allocation for bug assignment is one of the most important steps in software quality maintenance. Team allocation is generally done from previously fixed reports. Due to ignoring recent activities, these approaches may allocate inactive fixers. Both previous reports and recent commits do not contain any information regarding the newly joined developers. As a result, these techniques fail to assign task to new developers. Not considering new developers in the final suggestion leads to improper task allocation. To overcome these limitations, firstly an expertise and recency based bug assignment technique is proposed, which combines both the current and past activity information to accurately suggest developers. Next, utilizing this concept a novel team allocation technique has been devised, which ensures bug assignment to both existing and new developers. This chapter first discusses the combining procedure of source commits and past reports in automatic bug assignment along with the reasoning of the obtained results. Next, the proposed team allocation technique is discussed with its achievement. Afterwards, the possible threats of the proposed approach is presented. Finally, this chapter concludes the thesis by outlining the possible future directions.

6.1 Discussion

In order to assign tasks to both existing and new developers, a team allocation technique is proposed in this thesis. The technique suggests a recent and expert team of developers by considering both existing and new developers. In the following Subsections, the proposed team assignment technique and its compatibility analysis are summarized.

6.1.1 Expertise and Recency Based Bug Assignment

Analysing software source code related commits for identifying recent developers is a common practice in bug assignment [74]. Although this approach can recommend active developers, it totally ignores the developers' expertise in fixing similar bug reports. Due to considering only recent source commits, this approach can recommend novice or inexperienced developers. In order to assess an appropriate fixer for a newly arrived bug, both experienced and recent developer needs to be considered. So, an Expertise and Recency based Bug Assignment approach known as ERBA is proposed. These techniques take source code as inputs. It then extracts the name of source identifiers from each source code files. An index is constructed that maps the identifiers to the developers who used those terms along with the usage time. This index represents recent activities of developers. Besides, keywords are also extracted from the *summary* and *description* property of each past report. These keywords are then mapped to the developers who fixed the report to produce another index denoting the past activity of developers. For new bug reports, these indexes are searched with the keywords of the new report. The search results are then combined using tf-idf term weighting technique to assign a score which represents expertise and recency of developers.

ERBA was applied on three open source projects - Eclipse JDT, AspectJ

and SWT to assess its compatibility. The results are compared with a source commit recency based technique, ABA-time-tf-idf and a bug history based expertise technique, TNBA. The results are evaluated using three metrics - Top N Rank or Accuracy, Effectiveness and Mean Reciprocal Rank (MRR). The results show that ERBA obtains the actual developers at top position of the list than the two exiting techniques. For example, ERBA retrieves about 98.18% of the fixers accurately, whereas ABA-time-tf-idf and TNBA retrieves only 43.66% and 97% respectively. The reason behind the lower accuracy of ABA-time-tf-idf is that it ignores the bug fixing history of developers. The combination of both ensures bug assignment to both active and expert developers. That is why, ERBA achieves higher Top N Ranking. This experimentation was further extended to analyse the effectiveness and MRR of the suggested fixer list. ERBA shows the first relevant developers near position 2.05, 2.02 and 2.5 for Eclipse JDT, Netbeans and AspectJ respectively. These lower values are achieved because ERBA considers the date of fixing and committing changes while determining the score of developers. As a result active developers are placed at the top of the lists, thus indicating higher effectiveness by ERBA. It also achieves improvement on MRR values of ABA-time-tf-idf and TNBA. For example - 3%, 1% and 12% improvements on MRR value are gained against TNBA for the three projects. The reason is that, incorporating the recent source commits in fixing history helps to less prioritize the inactive developers.

6.1.2 Team Allocation Considering New Developers

Subsection shows the importance of combining bug fixing experience with source commit activities in bug assignment. However, none of these information sources contain trace about joining and bug solving preference of new developers. As a result, existing techniques fail to allocate tasks to new developers. Moreover, the collaborative nature of bug fixing raises the need for team assignment.

Concerned with the above mentioned issues, a Team Allocation technique for ensuring task allotment to Existing and New (TAEN) developers has been developed. This technique first determines the bug type of each fixed bug reports using Latent Dirichlet Allocation (LDA) techniques. The developers who have fixed similar types of bugs are then collected to form developer groups correspondent to each bug type. As the technique focuses to suggest a team, a developers' collaboration network (composed of four types of nodes and eight types of edges) is constructed for identifying developers communication on solving bugs. The current load of developers is also determined using the unfixed bug reports. The new developers are then presented with the top likely keywords and bug reports of each bug type. These developers are then initially grouped under the chosen type(s). When new bug report arrives, its type is determined in a similar way and the grouped developers corresponding the type is selected as initial fixer group. The collaborations among this fixer group over the network is derived, and a score is assigned to developers based on the frequency and recency of the collaborations. Based on the severity of the new report, a team of existing and new developers is then suggested, using the assigned score and current workload of developers.

Experiments have been performed on three open source projects - Netbeans, Eclipse JDT, and AspectJ. The experimental results are evaluated using two metrics - Recall and Workload Allocation. The results are compared with an existing technique known as KSAP [4]. The results show that TAEN achieves recall of 78.55%, 67.18% and 61.27% on placing the actual fixers in the suggested teams, which are higher than KSAP (65.88%, 55.51% and 50.17% respectively. The reason behind achieving this higher recall is, TAEN applies the concept of recency and expertise combination mentioned in Subsection 6.1.2. The consideration of recent collaborations, enabled TAEN to retrieve more relevant developers. While scoring the developers, the workload assigned to the developers by TAEN and

KSAP are also measured. The results show that, TAEN is successful when allocating task to new developers. On the contrary, the existing technique totally ignores new developers. Collecting the bug solving preference allow TAEN to include the new developers in team suggestion. Besides, checking the *severity* of new reports while assigning teams, ensures task assignment to all developers which is ignored by KSAP.

A workload balancing approach was also employed to analyse the task allocation among developers. The results show that balancing workload and suggesting appropriate developers at the same time, is a trade-off. This goal can be achieved on real life scenarios, if applied from the scratch of the project. Because all the developers would get task from the beginning and would have expertise on different topics.

6.2 Threats to Validity

This section discusses the threats which can affect the validity of TAEN. The threats are identified from three perspectives - internal threats, external threats and construct threats.

- **Internal Threats:** The internal threat refers to the threats that affect the validity of the results. The implementation of TAEN and the environmental set up may affect the obtained results. The proposed technique as well as the experimental projects are implemented in java programming language. Therefore, the result gained through analysing the experimental projects may differ when experimented in platforms other than java. Although the technique is implemented in java, the proposed technique is language independent. Thus, it can be applied on other platforms as well.

- **External Threats:** The experimental projects that are chosen and the collected source repository, commit logs and bug reports may affect the

degree to which the results can be generalized. Bug reports are the most important elements from which the developers information are tended to be extracted [75]. So, the quality of the bug report plays important role in bug assignment. Non-informative bug reports may slow down the triaging and resolution process. If proper information are not provided, the developers pause working in the report and leave it by tagging "WorksForMe" as mentioned in Chapter 2, Subsection 2.1.2. For example, the recency information of developers' are derived by matching the new report keywords against the source identifiers. If the name of entities are not correctly used in the bug report, it may affect the recency collection procedure of ERBA. However, the keywords of the bug reports go through few pre-processing steps to remove redundant information from the report.

The commit logs extracted from the repository should contain information of the changed files. Otherwise the identifier indexing may be incomplete and thus recency determination may fail. The bug reports should contain the actual feature of the bug, otherwise bug type determination using LDA modeling can be affected. Moreover, all the investigation data are collected from open source community. Since only open source projects are used , the results might not be generalizable to closed-source projects. The reason is that for applying the technique substantial amount of bug reports, commits are required. However, if a closed-source project maitains enough records, ERBA and TAEN can be applied without difficulty. Another reason for choosing the projects is, those are used as study objects in most of the existing bug assignment techniques [1, 4, 6, 13].

- **Construct Threats:** Construct threats are related to the parameters and metrics which are used to analyse the effectiveness of the proposed technique. The experimental setup of TAEN divides the fixed bug reports

of Eclipse JDT, Netbeans and AspectJ into 17, 30 and 17 types respectively, A change in these parameter values may affect the recall of the technique. However, these values can be set using established techniques ([76–78]) and the bug triager as well. The results are analysed using metrics - Top N Rank [1], Effectiveness [1], Mean Reciprocal Rank (MRR) [1], Recall [4] and Workload Distribution. Therefore, analysing the results with other metrics can affect the generalization of the results. However, most of the considered metrics are widely applied in evaluation of bug assignment techniques.

6.3 Future Direction

In this thesis, a team allocation technique for ensuring bug assignment to new developers has been proposed. However, there remain a number of future scopes for improving the technique. The following points outline the potential future directions related to this thesis.

- Collecting the preference of new developers can be performed in a more structured way. Structural forms can be introduced to collect bug solving preference. Besides, curriculum vitae, open source profile links can be used to enrich the preference collection of new developers.

- Previously unseen bug reports can be incorporated. The reports which have not been reported earlier or solved by developers, needs to be handled.

- The existing techniques tend to assign tasks to developers who can fix it best. This assumption enables the techniques to achieve higher recall, but the developers may be over-utilized. Again, balancing tasks among developers may reduce the recall of bug assignment. Therefore, more research is required on the trade-off between recall and balanced workload allocation.

- TAEN is evaluated on widely used open source projects. So, the technique can be evaluated on industrial projects for observing its behaviour.

The proposed team allocation technique is devised to suggest appropriate fixers in the field of bug resolution. However, the concept of the technique can be applied on other fields as well. For example -

- Duplicate bug report detection techniques tend to identify similar bug reports by analysing links between the bug repository and commit history. So the technique of identifying similar keywords in both bug reports and commit logs of ERBA can be used in duplicate bug report detection [79,80].

- The concept of expertise and recency based team allocation can be used in task allocation to different fields such as Software Project Team Management, Employee Management etc ([81, 82]).

Bibliography

[1] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "A time-based approach to automatic bug report assignment," *Journal of Systems and Software*, vol. 102, pp. 109–122, 2015.

[2] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Improving automatic bug assignment using time-metadata in term-weighting," *IET Software*, vol. 8, no. 6, pp. 269–278, 2014.

[3] "Qa/bugzilla/fields/severity - the document foundation wiki," Jan. 2017. URL: `https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Severity` [accessed: 2017-01-10].

[4] W. Zhang, S. Wang, and Q. Wang, "Ksap: An approach to bug report assignment using knn search and heterogeneous proximity," *Information and Software Technology*, vol. 70, pp. 68–84, 2016.

[5] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, vol. 58, no. 2, pp. 1–24, 2015.

[6] H. Hu, H. Zhang, J. Xuan, and W. Sun, "Effective bug triage based on historical bug-fix information," in *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 122–132, IEEE, 2014.

[7] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *Proceedings of 34th International Conference on Software Engineering (ICSE)*, pp. 25–35, IEEE, 2012.

[8] G. Murphy and D. Cubranic, "Automatic bug triage using text categorization," in *Proceedings of the 6th International Conference on Software Engineering & Knowledge Engineering (SEKE)*, Citeseer, 2004.

[9] S. Banitaan and M. Alenezi, "Tram: An approach for assigning bug reports using their metadata," in *Proceedings of the 3rd International Conference on Communications and Information Technology (ICCIT)*, pp. 215–219, IEEE, 2013.

[10] O. Baysal, M. W. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *Proceedings of the 17th International Conference on Program Comprehension (ICPC)*, pp. 297–298, IEEE, 2009.

[11] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?," in *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pp. 451–460, IEEE, 2012.

[12] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR)*, pp. 131–140, IEEE, 2009.

[13] J.-W. Park, M.-W. Lee, J. Kim, S.-W. Hwang, and S. Kim, "Cost-aware triage ranking algorithms for bug reporting systems," *Knowledge and Information Systems*, vol. 48, pp. 679–705, 2015.

[14] V. Dedík and B. Rossi, "Automated bug triaging in an industrial context," in *Proceedings of the 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 363–367, IEEE, 2016.

[15] P. Bhattacharya, I. Neamtiu, and C. R. Shelton, "Automated, highly-accurate, bug assignment using machine learning and tossing graphs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2275–2292, 2012.

[16] "Home :: Bugzilla :: bugzilla.org," Jan. 2017. URL: `https://www.bugzilla.org/` [accessed: 2017-01-10].

[17] E. Börger and A. Cisternino, *Advances in software engineering*. Springer, 2008.

[18] "Software bug - Wikipedia," Jan. 2017. URL: `https://en.wikipedia.org/wiki/Software_bug` [accessed: 2017-04-27].

[19] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, 2002.

[20] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pp. 263–272, IEEE/ACM, 2011.

[21] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proceeding of the International Conference on Software Maintenance (ICSM)*, pp. 346–355, IEEE, 2008.

[22] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: improving cooperation between developers and users," in *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, pp. 301–310, ACM, 2010.

[23] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, vol. 58, no. 2, pp. 1–24, 2015.

[24] T. DeMarco and T. Lister, *Peopleware: productive projects and teams*. Addison-Wesley, 2013.

[25] J. D. Herbsleb, H. Klein, G. M. Olson, H. Brunner, J. S. Olson, and J. Harding, "Object-oriented analysis and design in software project teams," *Human–Computer Interaction*, vol. 10, no. 2-3, pp. 249–292, 1995.

[26] J. Anvik, "Automating bug report assignment," in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pp. 937–940, ACM, 2006.

[27] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 23–32, IEEE, 2003.

[28] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *in Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pp. 121–124, ACM, 2008.

[29] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pp. 27–30, ACM, 2008.

[30] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*, p. 2, IEEE Computer Society, 2007.

[31] J. Yang, Y.-G. Jiang, A. G. Hauptmann, and C.-W. Ngo, "Evaluating bag-of-visual-words representations in scene classification," in *Proceedings of the International Workshop on Multimedia Information Retrieval (MIR)*, pp. 197–206, ACM, 2007.

[32] C. Silva and B. Ribeiro, "The importance of stop word removal on recall values in text categorization," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, vol. 3, pp. 1661–1666, IEEE, 2003.

[33] N. Alemayehu and P. Willett, "The effectiveness of stemming for information retrieval in amharic," *Program*, vol. 37, no. 4, pp. 254–259, 2003.

[34] J. A. Goldsmith, D. Higgins, and S. Soglasnova, "Automatic language-specific stemming in information retrieval," in *Proceedings of the Workshop of the Cross-Language Evaluation Forum for European Languages (CLEF)*, pp. 273–283, Springer, 2000.

[35] A. G. Jivani *et al.*, "A comparative study of stemming algorithms," *Int. J. Comp. Tech. Appl*, vol. 2, no. 6, pp. 1930–1938, 2011.

[36] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[37] D. Harman, "How effective is suffixing?," *Journal of the american society for information science*, vol. 42, no. 1, p. 7, 1991.

[38] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[39] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pp. 111–120, ACM, 2009.

[40] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *Proceedings of the 26th International Conference on Software Maintenance (ICSM)*, pp. 1–10, IEEE, 2010.

[41] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[42] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pp. 361–370, ACM, 2006.

[43] A. Aizawa, "An information-theoretic perspective of tf–idf measures," *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003.

[44] D. J. Hu, "Latent dirichlet allocation for text, images, and music," *University of California, San Diego. Retrieved April*, vol. 26, pp. 1–19, 2009.

[45] T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Topic-based, time-aware bug assignment," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–4, 2014.

[46] V. B. Sawant and N. V. Alone, "A survey on various techniques for bug triage," *International Research Journal of Engineering and Technology*, vol. 2, pp. 917–920, 2015.

[47] L. Chen, X. Wang, and C. Liu, "An approach to improving bug assignment with bug tossing graphs and bug similarities," *Journal of Software*, vol. 6, pp. 421–427, 2011.

[48] K. Aggarwal, T. Rutgers, F. Timbers, A. Hindle, R. Greiner, and E. Stroulia, "Detecting duplicate bug reports with software engineering domain knowledge," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 211–220, IEEE, 2015.

[49] W. Zou, Y. Hu, J. Xuan, and H. Jiang, "Towards training set reduction for bug triage," in *Proceedings of the 35th Annual Computer Software and Applications Conference (COMPSAC)*, pp. 576–581, IEEE, 2011.

[50] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu, "Towards effective bug triage with software data reduction techniques," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 1, pp. 264–280, 2015.

[51] H. Schütze, "Introduction to information retrieval," in *Proceedings of the International Communication of Association for Computing Machinery conference*, 2008.

[52] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 97–106, ACM, 2010.

[53] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proceedings of the 20th IEEE/ACM International Symposium on the Foundations of Software Engineering (FSE)*, p. 63, ACM, 2012.

[54] T. F. Bissyande, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere, "Empirical evaluation of bug linking," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 89–98, IEEE, 2013.

[55] K. P. Murphy, "Naive bayes classifiers," *University of British Columbia*, 2006.

[56] K. Schwaber and M. Beedle, "Scrum: Agile software development," 2002.

[57] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?," in *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*, pp. 1–6, IEEE Computer Society, 2007.

[58] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[59] J. I. Maletic, M. L. Collard, and A. Marcus, "Source code files as structured documents," in *in Proceedings of the 10th International Workshop on Program Comprehension*, pp. 289–292, IEEE, 2002.

[60] R. Durrett, *Probability: theory and examples*. Cambridge university press, 2010.

[61] T. Fritz, G. C. Murphy, and E. Hill, "Does a programmer's activity indicate knowledge of code?," in *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp. 341–350, ACM, 2007.

[62] R. V. Sangle and R. D. Gawali, "Auto bug triage a need of software industry," *International Journal of Engineering Science*, vol. 8668, 2016.

[63] "JDT Core Component - Eclipse," Jan. 2017. URL: `https://eclipse.org/jdt/core/` [accessed: 2017-01-10].

[64] "The AspectJ Project - Eclipse," Jan. 2017. URL: `https://eclipse.org/aspectj/` [accessed: 2017-01-10].

[65] "SWT: The Standard Widget Toolkit - Eclipse," Jan. 2017. URL: `https://www.eclipse.org/swt/` [accessed: 2017-01-10].

[66] M. Steyvers and T. Griffiths, "Probabilistic topic models," *Handbook of latent semantic analysis*, vol. 427, no. 7, pp. 424–440, 2007.

[67] "Welcome to NetBeans," Jan. 2017. URL: `https://netbeans.org/` [accessed: 2017-01-10].

[68] "Eclipse JDT Bug Repository," Jan. 2017. URL: `https://bugs.eclipse.org/bugs/buglist.cgi?bug_status=__closed__&content=code&list_id=16384200&order=Importance&query_format=specific` [accessed: 2017-01-10].

[69] "Bug List - NetBeans," Jan. 2017. URL: `https://netbeans.org/bugzilla/buglist.cgi?limit=25&order=Importance&product=platform` [accessed: 2017-01-10].

[70] "AspectJ Bugs - Eclipse," Jan. 2017. URL: `https://eclipse.org/aspectj/bugs.php` [accessed: 2017-01-10].

[71] "Afrina/TREN," Jan. 2017. URL: `https://github.com/Afrina/TREN/blob/master/TeamAssignMSTestProject/Data/TeamData/bug_data_2009_2015.xml` [accessed: 2017-01-10].

[72] "NetBeans Sources - Mercurial Overview," Jan. 2017. URL: `http://hg.netbeans.org/` [accessed: 2017-01-10].

[73] "AspectJ Source Repository," Jan. 2017. URL: `https://github.com/eclipse/org.aspectj` [accessed: 2017-01-10].

[74] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *Proceedings of the ACM symposium on Applied computing*, pp. 1767–1772, ACM, 2006.

[75] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 308–318, ACM, 2008.

[76] R. Arun, V. Suresh, C. Veni Madhavan, and M. Narasimha Murthy, "On finding the natural number of topics with latent dirichlet allocation: Some observations," *Advances in Knowledge Discovery and Data Mining*, pp. 391–402, 2010.

[77] J. Cao, T. Xia, J. Li, Y. Zhang, and S. Tang, "A density-based method for adaptive lda model selection," *Neurocomputing*, vol. 72, no. 7, pp. 1775–1781, 2009.

[78] E. Zavitsanos, S. Petridis, G. Paliouras, and G. A. Vouros, "Determining automatically the size of learned ontologies.," in *ECAI*, vol. 178, pp. 775–776, 2008.

[79] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pp. 461–470, IEEE, 2008.

[80] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 15–25, ACM, 2011.

[81] M. A. Meyer, "The dynamics of learning with team production: Implications for task assignment," *The Quarterly Journal of Economics*, vol. 109, no. 4, pp. 1157–1184, 1994.

[82] J. S. Loucks and F. R. Jacobs, "Tour scheduling and task assignment of a heterogeneous work force: A heuristic approach," *Decision Sciences*, vol. 22, no. 4, pp. 719–738, 1991.