Intelligent Software Bugs Localization, Triage and Prioritization

by

Rafi Almhana

A dissertation submitted in fulfillment of the requirements for the degree of Doctor of Philosophy (Computer and Information Science) in the University of Michigan-Dearborn 2021

DEARBORN

ISELab

Intelligent Software Engineering

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my advisor, Professor Marouane Kessentini, who profound belief in my abilities. I will never forget his helpful advice, insightful feedback, and unparalleled support.

I would also like to extend my deepest gratitude to my wife, Dalia Abboud, who always be on my side. I will never forget her endless support and patience that cannot be underestimated.

In addition, I would like to thank my parents for their wise counsel and unwavering support.

I would like to acknowledge my colleagues for their wonderful collaboration.

Finally, thanks to all the people who made this dissertation possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

FCFS	First Come First Serve
GA	Genetic Algorithm
hNSGA-II	One Step Multi-objective Formulation based on NSGA-II
HMOA	Hybrid Multi-objective Approach
HS	History Similarity
LS	Lexical Similarity
MOSA	Mono Objective Simulated Annealing
NSGA-II	Non-dominated Sorting Genetic Algorithm

ABSTRACT

One of the time-consuming software maintenance tasks is the localization of software bugs especially in large systems. Developers have to follow a tedious process to reproduce the abnormal behavior then inspect a large number of files in order to resolve the bugs. Furthermore, software developers are usually overwhelmed with several reports of critical bugs to be addressed urgently and simultaneously. The management of these bugs is a complex problem due to the limited resources and the deadlines-pressure. Another critical task in this process is to assign appropriate priority to the bugs and eventually assign them to the right developers for resolution.

Several studies have been proposed for bugs localization, the majority of them are recommending classes as outputs which may still require high inspection effort. In addition, there is a significant difference between the natural language used in bug reports and the programming language which limits the efficiency of existing approaches since most of them are mainly based on lexical similarity. Most of the existing studies treated bug reports in isolation when assigning them to developers. They also lack the understanding of dynamics of changing bug priorities. Thus, developers may spend considerable cognitive efforts moving between completely unrelated bug reports. To address these challenges, we proposed the following research contributions:

1. We proposed an automated approach to find and rank the potential classes and methods in order to localize software defects. Our approach finds a good balance between minimizing the number of recommended classes and maximizing the relevance of the proposed solution using a hybrid multi-objective optimization algorithm combining local and global search. Our hybrid multi-objective

approach is able to successfully locate the true buggy methods within the top 10 recommendations for over 78% of the bug reports leading to a significant reduction of developers' effort comparing to class-level bug localization techniques.

2. We proposed an automated bugs triage approach based on the dependencies between several open bug reports. We defined the dependency between two bug reports as the number of common files to be inspected to localize the bugs. Then, we adopted multi-objective search to rank the bug reports for programmers. The results show a significant time reduction of over 30% in localizing the bugs simultaneously comparing to the traditional bugs prioritization technique based on only priorities.

3. We performed an empirical study to observe and understand the changes in bugs' priority in order to build a 3-W model on *Why* and *When* bug priorities change, and *Who* performs the change. We conducted interviews and a survey with practitioners as well as performed a quantitative analysis large database of bugs reports. As a result, we observed frequent changes in bug priorities and their impact on delaying critical bug fixes especially before shipping a new release.

CHAPTER I

Introduction

1.1 Research Context

A software bug is a coding error that may cause abnormal behaviors and incorrect results when executing the software system *Bruegge and Dutoit* (2004). After identifying an unexpected behavior of the software project, a user or developer will report it in a document, called a bug report *Zimmermann et al.* (2010). Thus, a bug report should provide useful information to identify and fix the bug. The number of these bug reports can be large. For example, MOZILLA had received more than 420,000 bug reports *Bettenburg et al.* (2008). These reports are important for managers and developers during their daily development and maintenance activities including bug localization *Fischer et al.* (2003). The process of finding the relevant source code fragments (methods, classes, etc.) that need to be modified to fix the bug according to a bug report description is defined as bug localization *Wang and Lo* (2014a).

The bug report is supposed to contain all information to (i) describe the bug, (ii) how to reproduce the bug, (iii) end users affected by the bug, (iv) the version of the software affected by the bug, (v) comments or feedback from other developers on the bug. Also, bug report has attributes such as (i) bug priority, (ii) bug severity, (iii) information about the creator of the bug and the developer assigned to work on the bug, (iv) the component or package related to the bug, (v) the date time stamp of bug creation and last modifications, (vi) the status of the bug or resolution needed to happen to fix the bug.

Software maintenance involves, typically, localizing and fixing a large number of defects that arise during development and evolution of systems *Zhang et al.* (2016). Localizing these software defects is expensive and time-consuming process which typically requires highly skilled and knowledgeable developers of the system. The localization process includes a manual search through the source code of the project in order to localize a single bug at a time *Jones* (2008). Due to the large number of reported bugs in successful projects, it is critical to efficiently manage them to improve developers productivity and quickly localize and fix these bugs *Zou et al.* (2018).

Bug priority is one of most critical attribute which describes the urgency on fixing the bug and therefore the scheduling to resolve the bug in the system. Bug severity is the impact of the bug in the software and therefore it describes how severe the bug is affecting the end users and the functionality of the software. Bug priority is the main factor that helps bug triagers to rank and analyze their bug reports before assigning them to programmers. Thus, a lot of research works *Tian et al.* (2013); *Yang et al.* (2014); *Sharma et al.* (2012); *Kumari and Singh* (2018); *Tian et al.* (2015) have been done to study bug priority and predicate or recommend the appropriate priority of a bug.

1.2 Challenges Summary

A developer always uses a bug report to reproduce the abnormal behavior to find the origin of the bug. However, the poor quality of bug reports can make this process tedious and time-consuming due to missing information. To find the cause of a bug, developers are not only using their own knowledge to investigate the bug report but interact with peer developers to collect additional information. An efficient automated approach for locating and ranking important code fragments for a specific bug report may lead to improve the productivity of developers by reducing the time to find the cause of a bug *Fischer et al.* (2003).

The majority of existing bug localization studies are mainly based on lexical matching scores between the statements of bug reports and the name of code elements in software systems *Sun et al.* (2010); *Nguyen et al.* (2011); *Ashok et al.* (2009). However, there is a significant difference between the natural language used in bug reports and the programming language which limits the efficiency of existing approaches.

Although several techniques have been proposed to localize bugs *Wong et al.* (2016); *Almhana et al.* (2016) and predict the severity of bugs *Uddin et al.* (2017); *Chaturvedi and Singh* (2012); *Zhang et al.* (2016), the existing studies related to the management of bugs report are mainly based on the priority scores to rank and assign bug reports without looking to the possible dependencies between them *Zheng et al.* (2006); *Canfora et al.* (2011); *Li et al.* (2006). Thus, developers may get assigned bug reports related to completely different files to be inspected which may increase the cognitive effort of the developers navigating between these independent bug reports.

Assigning appropriate priority to bugs is critical for timely addressing important software maintenance issues. An underlying aspect is the effectiveness of assigning priorities: if priorities of a fair number of bugs are changed, it could indicate delays in fixing critical bugs. To the best of our knowledge, there has been little prior work on understanding the dynamics of changing bug priorities.

1.3 Research Objectives

- We will evaluate the current software bugs localization solutions in order to find solutions to bridge the gaps and make improvements. We will design a proposal for our solution and implement the suitable application to solve industrial challenges and to enable researchers for another research venue.

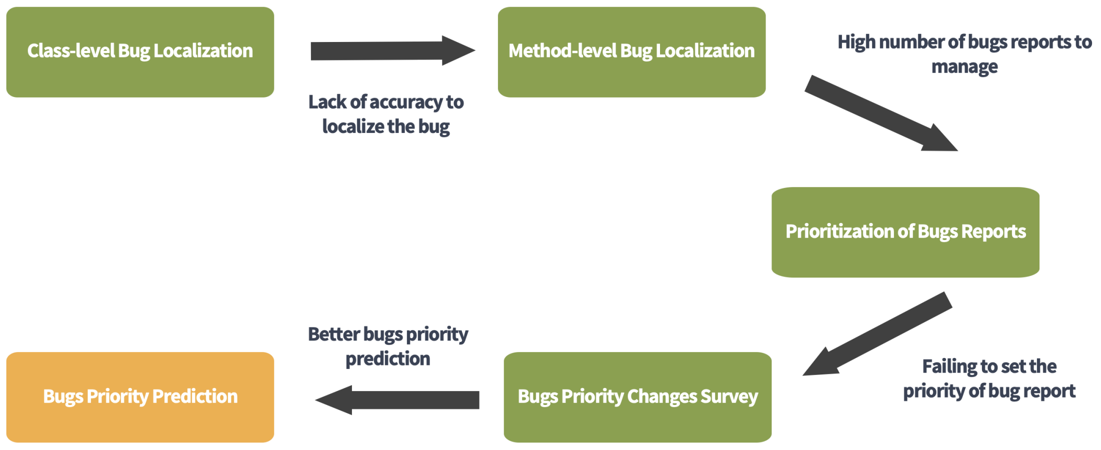

Class-level Bug Localization
Lack of accuracy to localize the bug
Method-level Bug Localization
High number of bugs reports to manage
Prioritization of Bugs Reports
Failing to set the priority of bug report
Bugs Priority Changes Survey
Better bugs priority prediction
Bugs Priority Prediction

Figure 1.1: Our research contributions

- We will evaluate the current related work about managing software defects' reports, study the pros and cons of the state of the art solutions, and discover the challenges needed to improve the process of managing the reports of software defects. We will design and implement a solution to fit industrial needs and open the door for areas of improvements.

- We will observe the dynamics in priority of software defects. We will study and detect the behavioral patterns and technical challenges come with changes in software defect's priority. We will conduct a survey to collect stakeholders' feedback. We will build our solution's model to recommend and present our findings. We will design and improve the state of the art mechanism in predicating defect's priority.

1.4 Proposed Contributions

To achieve our research objectives, we propose the research methodology as shown in Figure 1.1

1.4.1 Contribution 1: Method-Level Bug Localization Using Hybrid Multi-objective Search

Due to lack of details, large number of files to inspect, and high inspection efforts to to localize bugs in a software, we have initiated our research by proposing method-level bugs localization.

We propose an automated approach to find and rank the potential methods in order to localize the source of a bug based on a bug report description. Our approach finds a good balance between minimizing the number of recommended classes and maximizing the relevance of the proposed solution using a hybrid multi-objective optimization algorithm combining local and global search. The relevance of the recommended code fragments is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. Our approach operates on two main steps. The first step is to find the best set of classes satisfying the two conflicting criteria of relevance and the number of classes to recommend using a global search based on NSGA-II. The second step is to locate the most appropriate methods to inspect, using a local multi-objective search based on Simulated Annealing (MOSA) from the list of classes recommended by the first step. We evaluated our system on 6 open source Java projects, using the version of the project before fixing the bug of many bug reports. Our hybrid multi-objective approach is able to successfully locate the true buggy methods within the top 10 recommendations for over 78% of the bug reports leading to a significant reduction of developers' effort comparing to class-level bug localization techniques. The experimental results show that the search-based approach significantly outperforms four state-of-the-art methods in recommending relevant files for bug reports.

1.4.2 Contribution 2: Considering Dependencies Between Bug Reports to Improve Bugs Triage

Due to high number of open bugs reports to be addressed urgently and simultaneously, lack of efficient way to prioritize bugs reports and therefore provide a smooth transition for developers between multiple bugs reports. We propose multi-objective search to rank the bug reports for programmers based on both their priorities and the dependency between them.

We propose an automated bugs triage approach based on the dependencies between the open bug reports. Our approach starts by localizing the files to be inspected for each of the pending bug reports. We defined the dependency between two bug reports as the number of common files to be inspected to localize the bugs. Then, we adopted multi-objective search to rank the bug reports for programmers based on both their priorities and the dependency between them. We evaluated our approach on a set of open source programs and compared it to the traditional approach of considering bug reports in isolation based mainly on their priority. The results show a significant time reduction of over 30% in localizing the bugs simultaneously comparing to the traditional bugs prioritization technique based on only priorities.

1.4.3 Contribution 3: Understanding and Characterizing Changes in Bugs Priority: The Practitioners' Perceptive

Due to the importance of bugs priority in addressing critical bugs resolution, observing delays in fixing bugs and deploying software updates, and lack of prior work on understanding the dynamics of changing bug priorities. We conducted a survey with practitioners as well as performed a quantitative analysis on several bugs reports to better understand the dynamics of changing bug priorities and to enable researchers to build automated tools for checking and validating requests for bug priority changes.

We performed an empirical study to observe and understand the changes in bugs' priority in order to build a 3-W model on *Why* and *When* bug priorities change, and *Who* performs the change. We conducted interviews and a survey with practitioners as well as performed a quantitative analysis of X bugs reports, developers' comments, and source code changes from Y open source systems. The interviews with 11 developers from eBay aim to establish an initial model to characterize the changes in bugs priority. The survey with an additional 38 developers is to understand their experience in why and when bug priorities change, and who performs the change. Then, we conducted a manual inspection of the collected data on open source projects to compare our final bugs priority change model with changes identified in practice. Our quantitative results confirmed the outcomes of our interviews and surveys. For instance, we observed frequent changes in bug priorities and their impact on delaying critical bug fixes especially before shipping a new release. Our findings can enable (1) researchers to build automated tools for checking and validating requests for bug priority changes, (2) practitioners to use a standard format in documenting and approving bug priority changes, and (3) educators to teach the better management of bug priorities.

The above contributions led to different publications including:

- Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, Ali Ouni: Recommending relevant classes for bug reports using multi-objective search. ASE 2016 conference, acceptance rate 16% : 286-295, IEEE

- Almhana, R., Ferreira, T., Kessentini, M. and Sharma, T., 2020, September. Understanding and Characterizing Changes in Bugs Priority: The Practitioners' Perceptive. In 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 87-97), acceptance rate 24%. IEEE

- Almhana, R., Kessentini, M. and Mkaouer, W., 2020. Method-Level Bug Localization Using Hybrid Multi-objective Search. Information and Software Technology Journal, Volume 131, 32 pages, Elsevier, Impact Factor 2.73
- Almhana, R. and Kessentini, M., 2020. Detecting Dependencies Between Bug Reports to Improve Bugs Triage. Automated Software Engineering Journal, 26 pages, to appear, Impact Factor 1.97

1.5 Organization of the Dissertation

This dissertation is organized as follows: Chapter II introduces the current state of the art and related works to this dissertation. Chapter III presents method-level bug localization using hybrid multi-objective search. Chapter IV describes our proposed approach to improve bugs triage considering dependencies between bug reports. Chapter V describes our proposed model for understanding and characterizing changes in bugs priority, constructed from practitioners' perspective. Finally, a summary and future research directions are presented in VI.

CHAPTER II

State of the Art

2.1 Introduction

In this chapter, we cover the necessary background information related to our work followed by an overview of existing studies *Kessentini et al.* (2010); *Ghannem et al.* (2011); *Kessentini et al.* (2013a,b); *Ghannem et al.* (2014); *Mansoor et al.* (2015); *Almhana et al.* (2016); *Kessentini and Ouni* (2017); *Alizadeh et al.* (2018); *AlOmar et al.* (2019). We classified existing work in three main categories: 1) Bug localization; 2) Bug reports management and 3) Bug's priority.

2.2 Bug Localization

The problem of bug localization can be considered as searching the source of a bug given its description. To address this problem, the majority of existing studies is based on the use of Information-Retrieval (IR) techniques through the detection of textual and semantic similarities between a newly given report and source code entities *Sun et al.* (2010). Several IR techniques have been investigated, namely the Latent Semantic Indexing (LSI) in work of Dumais et al. *Dumais* (2004), Latent Dirichlet Allocation (LDA) in work of Blei et al. *Blei et al.* (2003) and the Vector Space Model (VSM) in work of Salton et al. *Salton et al.* (1975). Also, hybrid models extracted from these IRs techniques to tackle the problem of bug localization were proposed *Ye et al.* (2014).

We summarize, in the following, the different tools and approaches proposed in the literature based on the above IR techniques. BugScout in work of Nguyen et al. *Nguyen et al.* (2011) is a topic-based approach using LDA to analyze the bug related information (description, comments, external links, etc.) to detect the source of a bug and duplicated bug reports. The main limitation of BugScout is the dependency of the results on the keywords entered by the user. DebugAdvisor's in work of Ashok et al. *Ashok et al.* (2009) is a bug investigation system that takes as input a bug report in terms of text queries then uses them to mine existing fixed bug repository and generate a graph of possible reports. However, DebugAdvisor accuracy depends on the accuracy of the report's description and its accuracy when describing the bug and its related code entities.

BugLocator in work of Zhou et al. *Zhou et al.* (2012) combines several similarity scores from previous bug reports for bug localization. It generates a VSM model to extract suspect source files for a given bug report. Then, BugLocator mines previously fixed bug reports along with related files involved to rank suspect code fragments. The main issue raised in this work is the proneness of the weight density to the noise in the large files. To overcome this limitation, the work of Wong et al. *Wong et al.* (2014) added segmentation and stack trace analysis to improve the performance of the BugLocator approach. The limitation of this extension is that execution traces are not necessarily available in bug repositories.

BLUiR, the work of Saha et al. *Saha et al.* (2013) has been proposed also to compare a bug report to the structure of source files. It decomposes reports into summaries and then uses the structural retrieval to calculate similarities between these tokenized elements and source code ones to rank source code files. Saha et al. *Saha et al.* (2014) extended BLUiR to consider similar reports information, similarly to BugLocator as an additional similarity score. DHbPd in the workf of Rao et al. *Rao and Kak* (2011) incorporated code change information for bug localization. The main idea is to consider recently changed source code elements as potential candidates for hosting a bug.

Ye et al. *Ye et al.* (2014) have modeled the similarity between bug reports and source code through several characteristics that are captured through the use of 6 similarity features that describe the projects' domain knowledge. The combination of these measures is fed to a ranking heuristic called learning-to-rank. The ranking model returns the top candidate source files to investigate for a given bug report. The main originality of their work is the use of project's API description and auto-generated documentation as one of the features to utilize to reduce the lexical gap between the human description and the source code.

Ye et al. *Ye et al.* (2016a) extended their previous work by extending their ranking features utilized by learning-to-rank from 6 to 19. Besides the existing surface lexical similarity, API-based lexical similarity, collaborative filtering, code elements naming similarity, fixed bugs frequency, they included other source code characteristics that can be extracted from the projects such as summaries, naming conventions, interclass dependencies, etc. Although taking these features into account has given better results in terms of better files ranking, such information may not be available in all projects and sometimes it may be outdated and that may deteriorate the localization accuracy.

AmaLgam *Wang and Lo* (2014b) introduced the aggregation of relevant similarities extracted from source files, version history data and previously resolved bugs to calculate a global score for ranking files. This approach has given promising results compared to the previous techniques as it does not only combines code structure with previous reports but also it involves historical data to maximize the bug information coverage and enhance the localization accuracy.

Lamkanfi et al. developed a binary classifier to determine whether or not a bug is severe*Lamkanfi et al.* (2010). The report features were used as a training set to conduct a comparison between several classifiers, namely SVM, Naïve Bayes, Multinomial Naïve Bayes, and Nearest Neighbor. The experiments have shown that these classifiers outperform random severity assignment formulations. Similarly, the work of Lo et al. *Tian et al.* (2013) presented a classification engine labeled GRAY that extends the linear regression to predict the priority of bugs, but not bugs localization, while taking into account various external and internal report characteristics, extracted as features, then used to train the model.

Table 2.1 shows the most recent and relevant studies to our approach. It illustrates the differences among those studies in terms of input, output, and technique used to solve the bug localization problem. We notice that among all the studies listed in Table 2.1, the majority of them are related to recommend classes using Information Retrieval (IR) techniques. There are four different studies that addressed method-level bug localization. In the work of Youm et al. *Youm et al.* (2017), the authors utilized bug report description with code changes, source code, comments and stack traces and developer's log to find relevant methods.

Another approach is proposed by Lukins et al. *Lukins et al.* (2010) to localize bugs at the methods level using a static Latent Dirichlet allocation (LDA) technique which is solely based fon source code. In another study *Ye et al.* (2016a), Ye et al. used learning to rank technique to develop a ranking model in which they assign a weight for each source code file as a result of several features such as source code, bug description, code changes history and bugs-fixing history. The proposed approach can be adopted for both class and method levels. We used this approach as one of the baselines to evaluate the performance of our approach since the authors provided a replication package.

Study	**Input**	**Output**	**Technique**	**Date**	**Category**
Huang, Qiao, et al. *Huang et al.* (2017)	Bug Report, Source code	Package	IR, ML	Oct 2017	Information Retrieval
Wen, Ming, et al. *Wen et al.* (2016)	Developer's log, Software Changes.	Class	IR	Sep 2016	
Youm, Klaus Changsun, et al. *Youm et al.* (2017)	Comments, Stack Traces, Developer's Log and Code Changes.	Method	IR	Feb 2017	
Lukins, Stacy K., et al. *Lukins et al.* (2010)	Source Code	Method	IR (LDA)	Sep 2010	
Tantithamthavorn, Chakkrit, et al. *Tantithamthavorn et al.* (2018)	Source Code, Bug Report	Method	IR-based Classifier	Oct 2018	
Ye, Xin, et al. *Ye et al.* (2015)	Source Code, Bug Description, Code Changes	Method	Ranking Model	Sep 2015	Ranking Model
Pablo Loyola, et al. *Loyola et al.* (2018)	Code Changes	Class	Ranking Model	Oct 2018	
An Ngoc Lam, et al. *Lam et al.* (2017)	Source code, Bug Report	Class	IR (rVSM), neural network	May 2017	Neural Network
Yan Xiao, et al. *Xiao et al.* (2018)	Bug Report	Class	deep learning translation	July 2018	
Almhana, Rafi, et al. *Almhana et al.* (2016)	Source code, Bug Report, History of Bug Report	Class	Search Based Software Engineering	Sep 2016	Search Based

Table 2.1: Recent Studies

2.3 Bugs Reports Management

A survey on bug prioritization was proposed in *Uddin et al.* (2017). The authors collected 84 papers about bug prioritization or related topics from 2000 to 2015, they eliminated 32 papers after 2 steps review process. The majority of those papers used information retrieval technique such as Naive Bayes, Support Vector Machine (SVM) and Neural Networks for bugs prioritization. The survey focused mainly on predicting bugs priority and to estimate the severity of the bugs.

Table 2.2 summarizes the main studies related to bugs management and prioritization.

Kanwal et al. *Kanwal and Maqbool* (2012) proposed a classification based approach to develop a tool which uses the Naive Bayes and Support Vector Machine (SVM) classifiers. This tool mines the bug data from a bug repository so that it builds a piece of knowledge about the software to be inspected and its bugs repository and eventually rank or classify bugs.

The authors in *Alenezi and Banitaan* (2013) proposed an approach to predict the priority of bug report using different machine learning algorithms like Naive Bayes, Decision Trees, and Random Forest.

Xuan et al. *Xuan et al.* (2012) proposed a new way to prioritize bugs based on 3 different stages from mining the social interactions between developers.

Search-Based Software Engineering (SBSE) uses a computational search approach to solve optimization problems in software engineering *Harman and Jones* (2001). Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, fitness function, and solution change operators, there is a multitude of search algorithms that can be applied to solve that problem. Many search-based software testing techniques have been proposed for test cases generation *Núñez et al.* (2013), mutation testing *Henard et al.* (2014), regression testing *Shelburg et al.* (2013) and testability transformation. However, the problem of bugs localization

Study	**Input**	**Output**	**Technique**	**Published**
Yu et al. *Yu et al.* (2010)	Bug reports	Predict bug priority	Neural Networks	2010
Jaweria Kanwal *Kanwal and Maqbool* (2010)	Bug reports	Recommend bug priority	SVM	2010
Lamkanfi et al. *Lamkanfi et al.* (2011)	Bug reports	Predict the severity of bug	Naive Bayes	2010
Chaturvedi and Singh *Chaturvedi and Singh* (2012)	Bug reports	Determine bug severity	Naive Bayes	2012
Abdelmoez et al. *Abdelmoez et al.* (2012)	Bug reports	Predict bug fix-time	Naive Bayes	2012
Dommati et al. *Dommati et al.* (2013)	Bug reports	Classify bug reports	Naive Bayes	2012
Kanwal and Maqbool *Kanwal and Maqbool* (2012)	Bug reports	Prioritize bug reports	SVM	2012
Sharma et al. *Sharma et al.* (2012)	Bug reports	Predict bug priority	SVM	2012
Thung et al. *Thung et al.* (2012)	Bug reports	Predict bug priority	SVM	2012
Tian et al. *Tian et al.* (2012)	Bug reports	Predict the severity of bug	Nearest Neighbors	2012
Xuan et al. *Xuan et al.* (2012)	Develope prioritization	Predict the severity of bug	NB, SVM	2012
Alenezi and Banitaan *Alenezi and Banitaan* (2013)	Bug reports	Predict bug priority	Decision Tree, Random Forests	2013
Zanetti et al. *Zanetti et al.* (2013)	Bug reports	Classify bug reports	SVM	2013
Behl et al. *Behl et al.* (2014)	Bug reports	Predict the severity of bug	TF-IDF	2014
Garcia and Shihab *Valdivia Garcia and Shihab* (2014)	Bug reports	predicting blocking bugs	Decision trees	2014
Goyal et al. *Goyal et al.* (2015)	Bug reports	Predict bug priority	Bayes Net, Random Forest,	2015

Table 2.2: Overview of bug prioritization related work

was not addressed before using SBSE. The closest problem addressed using SBSE techniques is the bugs prioritization problem *Dreyton et al.* (2015). A mono-objective genetic algorithm was proposed to find the best sequence of bugs resolution that maximizes the relevance and importance of the bugs to fix while minimizing the cost. The main limitation of this work is the use of a mono-objective technique that aggregates two conflicting objectives. To overcome the limitation of aggregating two attributes that may experience conflicts, they extended their work *Dreyton et al.* (2016) to better find the trade-off between bugs with low relevance to the users may have high severity scores.

The problem of bug localization can be considered as searching the source of a bug given its description. To address this problem, the majority of existing studies is based on the use of Information-Retrieval (IR) techniques through the detection of textual and semantic similarities between a newly given report and source code entities *Sun et al.* (2010). Several IR techniques have been investigated, namely the Latent Semantic Indexing (LSI) *Dumais* (2004), Latent Dirichlet Allocation (LDA) *Blei et al.* (2003) and the Vector Space Model (VSM) *Salton et al.* (1975). Also, hybrid models extracted from these IRs techniques to tackle the problem of bug localization were proposed *Ye et al.* (2014).

2.4 Bug's Priority

Most of existing defect management studies focused on the prediction of bug severity/priority from bug reports *Tian et al.* (2013, 2015); *Yang et al.* (2014); *Sharma et al.* (2012); *Alenezi and Banitaan* (2013). Machine learning algorithms were extensively used for that purpose such as Support Vector Machine, Naive Bayes, K-Nearest Neighbors and Neural Networks. To the best of our knowledge, there is no existing study about understanding the changes in bug's priority and their rationale. We present, in the following, the closest studies to this thesis but a more comprehensive summary can be found in Table 2.3 about the prediction of bugs' priority.

Yang et al. *Yang et al.* (2014) proposed an approach to manage the bug triage by predicting the workload. They were also able to extract and identify multi-feature (e.g., Component, product, priority and severity) from bug report in order to assign developers to bugs and predict severity of those bugs *Yang et al.* (2014).

Tian et al. *Tian et al.* (2013, 2015) proposed an automated approach using machine learning to recommend a priority level based on information available in bug reports. Their method used several factors such as temporal, textual, author, related-report, severity, and product, to predict the priority level of a bug report *Tian et al.* (2013, 2015).

In the work of Sharma et al. *Sharma et al.* (2012), they use different machine learning techniques such as Support Vector Machine, Naive Bayes, K-Nearest Neighbors and Neural Network in predicting the priority of bugs. Also, they evaluated the performance by performing cross project validation *Sharma et al.* (2012). Similarly, Kumari et al. *Kumari and Singh* (2018) built classifiers using machine learning and Naïve Bayes and Deep Learning techniques. These classifiers considered the severity, summary weight and entropy attribute to recommend the priority of bugs *Kumari and Singh* (2018).

Yu et al. *Yu et al.* (2010) used neural network techniques to predict the priorities of bugs, adopted evolutionary training process to solve problems associated with reducing features, and reused data sets from similar software systems to speed up the convergence of training *Yu et al.* (2010).

Kanwal et al. *Kanwal and Maqbool* (2012), proposed a priority recommendation module based on Naïve Bayes and Support Vector Machine. Also, they provided another comparative study to evaluate which classifier performs better in terms of accuracy *Kanwal and Maqbool* (2012).

Alenezi et al. *Alenezi and Banitaan* (2013) presented an approach to predict the priority of a reported bug using different machine learning algorithms namely Naive Bayes, Decision Trees, and Random Forest. They also evaluated the performance of each one of these algorithms in predicting the priority of bug reports *Alenezi and Banitaan* (2013).

As a summary, all existing research papers focus on predicting the priority of bugs and therefore it helps in the bugs triage process and assigning developers to given bugs. More details can be found in Table 2.3.

Study	**Description / Technique Used**	**Have they addressed priority changes?**
Yang et al. *Yang et al.* (2014)	Extract and identify multi-feature (e.g., Component, product, priority and severity) from bug report	No
Tian et al. *Tian et al.* (2013, 2015)	Use several factors such as temporal, textual, author, related-report, severity, and product, to predict the priority level of a bug report	No
Sharma et al. *Sharma et al.* (2012)	Use Support Vector Machine, Naive Bayes, K-Nearest Neighbors and Neural Network in predicting the priority of bugs	No
Kanwal et al. *Kanwal and Maqbool* (2010, 2012)	Propose a priority recommendation module based on Naïve Bayes and Support Vector Machine.	No
Yu et al. *Yu et al.* (2010)	Utilize neural network techniques to predict the priorities of bugs	No
Alenezi et al. *Alenezi and Banitaan* (2013)	Present an approach to use different machine learning algorithms namely Naive Bayes, Decision Trees, and Random Forest	No
Kumari et al. *Kumari and Singh* (2018)	Build classifiers using machine learning and Naïve Bayes and Deep Learning techniques	No

Table 2.3: Summary of previous studies about bug priority predictions.

CHAPTER III

Method-Level Bug Localization Using Hybrid Multi-objective Search

3.1 Introduction

A software bug is a coding error that may cause abnormal behaviors and incorrect results when executing the system *Bruegge and Dutoit* (2004). After identifying an unexpected behavior of the software project, a user or developer will report it in a document, called a bug report *Zimmermann et al.* (2010). Thus, a bug report should provide useful information to identify and fix the bug. The number of these bug reports can be large. For example, MOZILLA had received more than 420,000 bug reports *Bettenburg et al.* (2008). These reports are important for managers and developers during their daily development and maintenance activities including bug localization *Fischer et al.* (2003). The process of finding the relevant source code fragments (methods, classes, etc.) that need to be modified to fix the bug according to a bug report description is defined as bug localization *Wang and Lo* (2014a).

A developer always uses a bug report to reproduce the abnormal behavior to find the origin of the bug. However, the poor quality of bug reports can make this process tedious and time-consuming due to missing information. To find the cause of a bug, developers are not only using their own knowledge to investigate the bug report but interact with peer developers to collect additional information. An efficient automated approach for locating and ranking important code fragments for a specific bug report may lead to improve the productivity of developers by reducing the time to find the cause of a bug *Fischer et al.* (2003).

The majority of existing bug localization studies are mainly based on lexical matching scores between the statements of bug reports and the name of code elements in software systems *Sun et al.* (2010); *Nguyen et al.* (2011); *Ashok et al.* (2009). However, there is a significant difference between the natural language used in bug reports and the programming language which limits the efficiency of existing approaches.

We considered, in this work, the following important observations. First, API documentation of the classes can be more useful than the name of code elements or comments to estimate the similarity between code fragments and bug reports *Aman et al.* (2019). Second, code fragments associated with previously fixed bug reports may be relevant also to the current report if these previously fixed bug reports are similar to a current bug report *Ye et al.* (2016b). Third, a code fragment that was fixed recently is more likely to still contain bugs than another class that was last fixed a long time ago *Zimmermann et al.* (2010). Fourth, a code fragment that has been frequently fixed, tend to be fault-prone and may cause more than one abnormal behavior in the future *Liblit et al.* (2003). Finally, the recommendation of a large number of classes to inspect may make the process of finding the cause of a bug time-consuming.

To address some of these challenges, we proposed in our previous work *Almhana et al.* (2016) a comprehensive approach for bugs localization based on bug reports description. We utilized a multi-objective optimization algorithm *Deb et al.* (2002) to find a balance between maximizing lexical and history-based similarity, and minimizing the number of recommended classes. The problem is formulated as a search for the best combination and sequence of classes from all the classes of the system that optimize as much as possible the balance between the above two conflicting objectives. The main feedback received from the participants of our experiments is that the file/class level recommendations are still time-consuming to explore and they very much prefer a precise localization at the method level.

In this work, we extended our previous work*Almhana et al.* (2016) to provide method-level bug localization, instead of the class-level bug localization. Our approach optimizes the trade-off between minimizing the number of recommended classes and maximizing the relevance of the proposed solution using a hybrid multi-objective optimization algorithm combining local and global search. The relevance of the recommended code fragments is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. Our approach includes two main steps: finding the best set of classes satisfying the two conflicting criteria of relevance and the number of classes to recommend and locating the most appropriate methods to inspect in those classes. We accomplish the former using a global search based on NSGA-II and the latter using a local multi-objective search based on Simulated Annealing (MOSA)*Czyzżak and Jaszkiewicz* (1998).

We have executed an extensive empirical evaluation of 6 large open-source software projects with more than 22,000 bug reports in total based on an existing benchmark *Ye et al.* (2014). The experimental results show that the search-based approach significantly outperforms four state-of-the-art techniques in recommending relevant files for bug reports including our previous work at the class *Nguyen et al.* (2011); *Ye et al.* (2016a); *Zhou et al.* (2012); *Almhana et al.* (2016) and method *Ye et al.* (2016a) levels. In particular, our hybrid multi-objective approach can successfully locate the true buggy methods within the top 10 recommendations at the methods level for over 78% of the bug reports.

The primary contributions of this these can be summarized as follows:

- To the best of our knowledge and based on recent surveys *Harman et al.* (2012), the thesis proposes one of the first search-based software engineering approaches to address the problem of finding relevant code fragments for bug reports. The approach combines the use of lexical and history-based similarity measures to locate and rank relevant code fragments for bug reports while minimizing the number of recommended classes.

- We extended our previous work by proposing a new hybrid multi-objective formulation, using NSGA-II and Mono Objective Simulated Annealing (MOSA), that combines global and local search to localize bugs at the method level instead of the class level.

- The thesis reports the results of an empirical study with an implementation of our hybrid multi-objective approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing techniques for bugs localization at the method and class levels *Nguyen et al.* (2011); *Ye et al.* (2016a); *Zhou et al.* (2012) based on a benchmark of 6 open source systems *Almhana and Kessentini* (2020b). We also compared the results of our hybrid multi-objective approach with a mono-objective formulation to make sure that our objectives are conflicting and our previous work which based only on a global search.

The remainder of this chapter is as follows: Section 2 describes the proposed approach and the hybrid search algorithm. The evaluation of our approach and its results on some research questions are explained in Section 3. Finally, concluding remarks and future work are provided in Section 4.

3.2 Approach

We first present an overview of our hybrid multi-objective approach to identify and prioritize relevant methods for bug reports, and then we describe the details of our formulation.

3.2.1 Approach Overview

Our approach aims at exploring a large search space to find relevant methods, to inspect by developers, given a description of a bug report. The search space is determined not only by the number of possible method combinations to recommend but also by the order in which they are proposed to the developer. In fact, bug reports may require the inspection of more than one method to identify and fix bugs.

Due to this large search space of potential solutions to explore, we propose a heuristic-based optimization method including two main steps. The first step, based on a global search, operates on the classes level while the second one, based on a local search, operates on the methods level of selected classes after the first step. A local search is used in the second step due to the reasonable size of the search space that consists of the methods of identified classes. We note that the search space at the method level after the first filtering step at the class level is still a large search problem due to the typical large size of classes. We noticed in our previous work for bugs localization at the class level that each file might have on average over 20 methods per class.

The difference between the two search strategies is not related to the objective space but to the change operators and population size/generation. In global search, we use a population of several solutions at each iteration, and both crossover and mutation operators to create a significant perturbation of the solutions at each iteration. However, the local search is limited to one solution (not a population) at each iteration and only a limited change operator (mutation) to create a limited variation at each iteration when generating a new solution. The rationale behind the difference of both search strategies that the population and both change operators can help to explore a large search space to identify relevant solutions using the global search (all the classes of the systems). Once these relevant solutions are identified then a local search can be applied to a smaller search space (limited number of classes that may contain the bug) using one solution at each iteration and only a mutation operator. The local search is faster than running another global search due to the limited search space.

The general structure of our approach is sketched in Figure 3.1. It takes 5 inputs as follows:

- The source code of the project to be inspected,
- The API specifications,
- The description of the bug report,
- A list of previous bug reports of the project,
- The history of the applied changes in previous releases of the project.

Our approach generates as output, in the first step, a near-optimal sequence of ranked classes that maximizes the relevance to the bug report and minimizes the number of recommended classes. The list of identified classes for inspection can be checked by the developer, as an optional step, to further reduce the number of class recommendations as shown in Figure 3.2. Then, the second step is executed to generate as output a near-optimal sequence of ranked methods that maximize the relevance to the bug report and minimizes the number of recommended methods out of the classes identified in the first step.

Both heuristic-based optimization steps are formulated based on two main conflicting objectives. The first objective is the correctness function that includes two routines:

- Maximizing the Lexical similarity between recommended code fragments (e.g. classes for the first step and methods for the second step) and the description of the bug report (including the API and name of code elements similarity);
- Maximizing the history-based function score that includes the number of recommended code fragments that have been fixed in the past, recent changes introduced by the developers to these code fragments and similarities with previous bug reports.

The second objective is to minimize the number of code fragments to recommend.

It is obvious that those two objectives are conflicting since maximizing the relevance of recommended code fragments may lead to low precision and thus increase the number of recommended code fragments. Thus, we consider, in this thesis, the task of bugs localization as a hybrid multi-objective optimization problem. We used the non-dominated sorting genetic algorithm (Non-dominated Sorting Genetic Algorithm (NSGA-II)) as a global search for the class level *Deb et al.* (2002) and the multi-objective Simulated Annealing algorithm (MOSA) *Czyżżak and Jaszkiewicz* (1998) as a local search for the method level.

When comparing the relative fitness of generated solutions, both NSGA-II and MOSA utilize the idea of Pareto optimality using dominance as a basis for comparison as described in the following definitions [1,2]. The set of trade-off solutions is called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front. Hence, the output of NSGA-II and MOSA consists in approximating the entire Pareto front as described in the definition [3].

The definition of Pareto optimality states that x^* is Pareto optimal if no feasible vector exists that would improve some objectives without causing a simultaneous worsening in at least one other objective.

Definition 1: Pareto optimality

A solution $x^* \in \Omega$ is Pareto optimal if $\forall x \in \Omega$ and $I = \{1, ..., M\}$ either $\forall m \in I$ we have $f_m(x) = f_m(x^*)$ or there is at least one $m \in I$ such that $f_m(x) > f_m(x^*)$.

Definition 2: Pareto dominance

A solution $u = (u_1, u_2, ..., u_n)$ is said to dominate another solution $v = (v_1, v_2, ..., v_n)$ (denoted by $f(u) \prec f(v)$) if and only if $f(u)$ is partially less than $f(v)$. In other words, $\forall m \in \{1, ..., M\}$ we have $f_m(u) \leq f_m(v)$ and $\exists m \in \{1, ..., M\}$ where $f_m(u) < f_m(v)$.

In addition to Pareto Optimality and Pareto Dominance, we need to define Pareto Optimal set and Pareto Optimal Front.

In the following, we describe an overview of both algorithms, the solution representation, a formal formulation of the two objectives to optimize and the change operators.

3.2.2 NSGA-II

In this thesis, we adapted one of the widely used multi-objective algorithms called NSGA-II *Deb et al.* (2002). NSGA-II is a powerful global search method stimulated by natural selection that is inspired by the theory of Darwin. Hence, the basic idea of NSGA-II is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. The first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation. Then, a child population Q_0 is generated from the population of parents P_0

Definition 3: Pareto optimal set

For a given MOP $f(x)$, the Pareto optimal set is
$P^* = \{x \in \Omega | \neg \exists x' \in \Omega, f(x') \prec f(x)\}$.

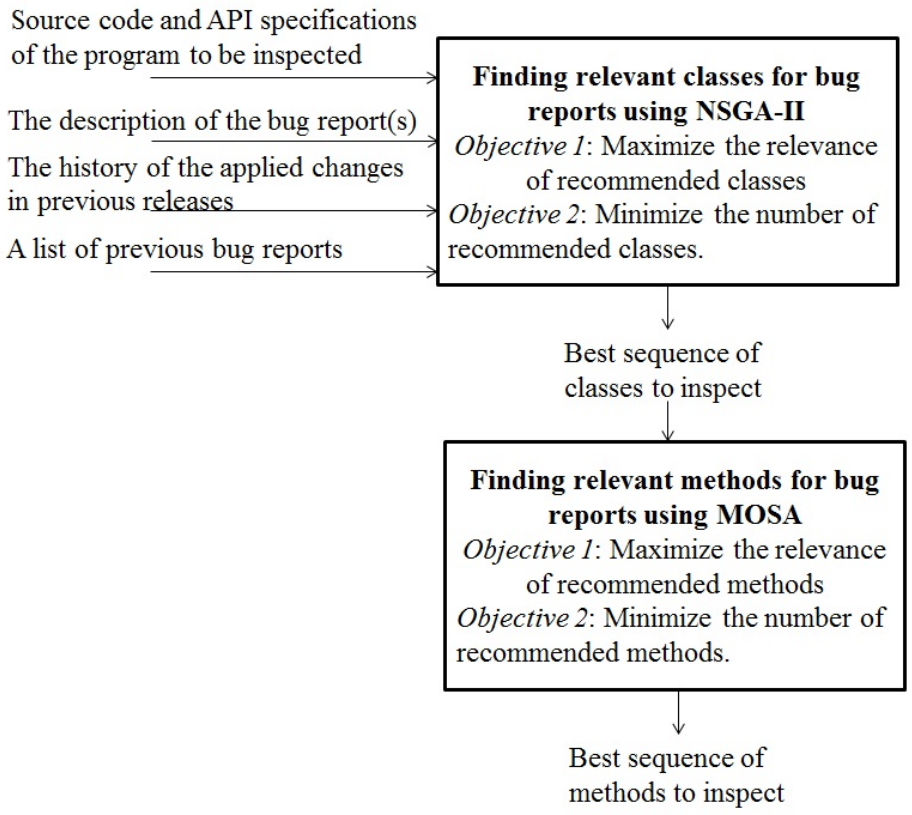
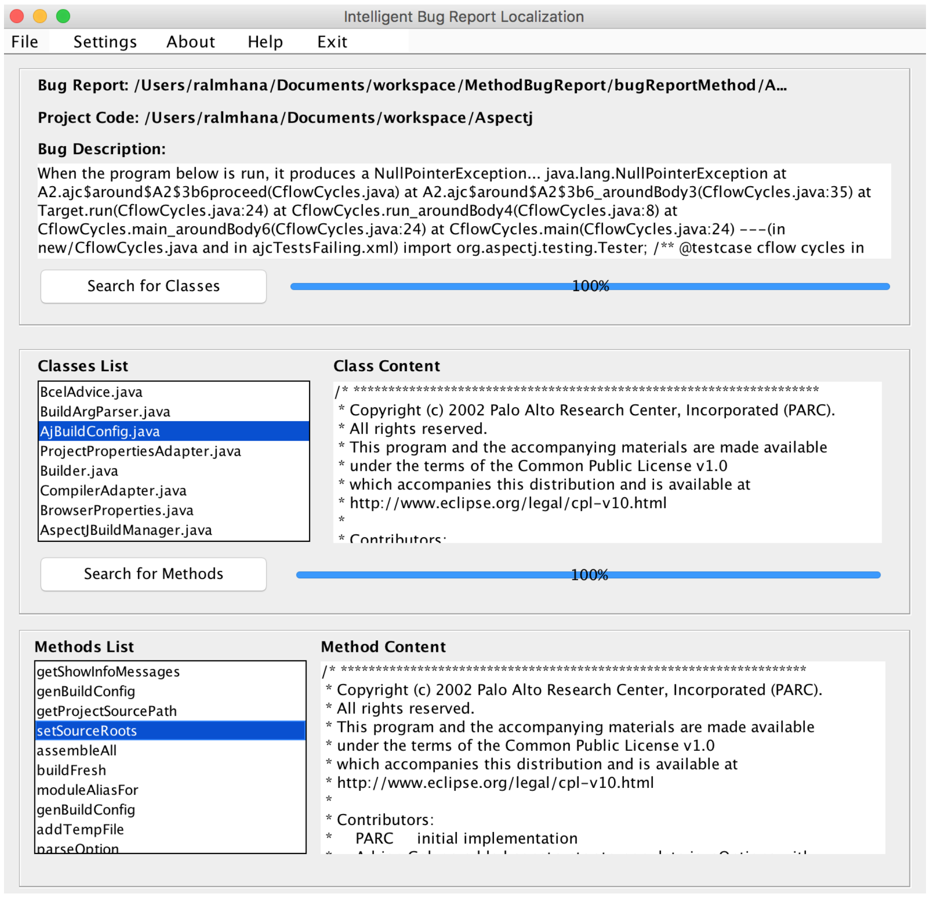

Source code and API specifications of the program to be inspected
The description of the bug report(s)
The history of the applied changes in previous releases
A list of previous bug reports
Finding relevant classes for bug reports using NSGA-II
Objective 1: Maximize the relevance of recommended classes
Objective 2: Minimize the number of recommended classes.
Best sequence of classes to inspect
Finding relevant methods for bug reports using MOSA
Objective 1: Maximize the relevance of recommended methods
Objective 2: Minimize the number of recommended methods.
Best sequence of methods to inspect

Figure 3.1: Approach Overview



Bug Report: /Users/ralmhana/Documents/workspace/MethodBugReport/bugReportMethod/A...

Project Code: /Users/ralmhana/Documents/workspace/Aspectj

Bug Description:

When the program below is run, it produces a NullPointerException... java.lang.NullPointerException at A2.ajc$around$A2$3b6proceed(CflowCycles.java) at A2.ajc$around$A2$3b6_aroundBody3(CflowCycles.java:35) at Target.run(CflowCycles.java:24) at CflowCycles.run_aroundBody4(CflowCycles.java:8) at CflowCycles.main_aroundBody6(CflowCycles.java:24) at CflowCycles.main(CflowCycles.java:24) ---(in new/CflowCycles.java and in ajcTestsFailing.xml) import org.aspectj.testing.Tester; /** @testcase cflow cycles in

Classes List

- BcelAdvice.java
- BuildArgParser.java
- AjBuildConfig.java
- ProjectPropertiesAdapter.java
- Builder.java
- CompilerAdapter.java
- BrowserProperties.java
- AspectJBuildManager.java

Class Content

```
/* *******************************************************************
 * Copyright (c) 2002 Palo Alto Research Center, Incorporated (PARC).
 * All rights reserved.
 * This program and the accompanying materials are made available
 * under the terms of the Common Public License v1.0
 * which accompanies this distribution and is available at
 * http://www.eclipse.org/legal/cpl-v10.html
 *
 * Contributors:
```

Methods List

- getShowInfoMessages
- genBuildConfig
- getProjectSourcePath
- setSourceRoots
- assembleAll
- buildFresh
- moduleAliasFor
- genBuildConfig
- addTempFile
- parseOption

Method Content

```
/* *******************************************************************
 * Copyright (c) 2002 Palo Alto Research Center, Incorporated (PARC).
 * All rights reserved.
 * This program and the accompanying materials are made available
 * under the terms of the Common Public License v1.0
 * which accompanies this distribution and is available at
 * http://www.eclipse.org/legal/cpl-v10.html
 *
 * Contributors:
 *     PARC     initial implementation
```

Figure 3.2: The proposed bugs localization tool.

using genetic operators such as crossover and mutation. Both populations are merged into an initial population R_0 of size N. As a consequence, NSGA-II starts by generating an initial population based on a specific representation that will be discussed later in Solution Representation section of this chapter. This initial population consists of a list of classes from the studied system. Thus, this population stands of a set of solutions represented as sequences of classes to inspect, which are randomly selected and ordered, for a specific bug report description taken as input.

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts. The dominance level becomes the basis of a selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions. When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance to make the selection. This front F_i to be split, is sorted in descending order, and the first $(N - |P_{t+1}|)$ elements of F_i are chosen. Then a new population Q_{t+1} is created using selection, crossover, and mutation. This process will be repeated until reaching the last iteration according to stop criteria. The following three subsections describe more precisely our adaption of NSGA-II to the model change detection problem.

3.2.3 Multi-Objective Simulated Annealing (MOSA)

Multi-objective Simulated Annealing is a local search heuristic inspired by the concept of annealing in metallurgy where metal is heated, raising its energy and relieving it of defects due to its ability to move around more easily *Ulungu et al.* (1999). As its temperature drops, the metal's energy drops and eventually it settles in a more stable state and becomes rigid. The local search algorithm of the Simulated Annealing is very suitable for exploring reasonable search spaces in terms of the size like in our case *Ulungu et al.* (1999).

Definition 4: Pareto optimal front

For a given MOP $f(x)$ and its Pareto optimal set P^*, the Pareto front is $PF^* = \{f(x), x \in P^*\}$.

The first step of the MOSA algorithm is to initialize a total of five parameters: temperature parameter T_0, cooling factor α and cooling step N_{Step}, final temperature T_{Stop} and the maximum number of iteration N_{Stop}.

In MOSA, the mutated solution will be kept and used for the next iteration if it dominates or is in the same non-dominating front as the solution from the previous iteration. To determine the probability that the mutated solution dominated by the solution from the previous iteration will be kept and used for the next iteration of MOSA, there are several possible acceptance probability functions that can be utilized.

Since the previous works *Shelburg et al.* (2013); *Ulungu et al.* (1999) have noted that the average cost criteria yields good performance we have utilized this metric. The average cost criteria simply takes the average of the differences of each objective value between two solutions, i and j, over all objectives D, as shown in Equation 1. The final acceptance probability function used in MOSA is shown in Equation 2.

$$c(i,j) = \frac{\sum_{k=1}^{|D|} c_k(j) - c_k(i)}{|D|} \tag{3.1}$$

$$AcceptProb(i,j,temp) = e^{\frac{-abs(c(i,j))}{temp}} \tag{3.2}$$

Where *temp* is the current temperature and $c_k(i)$ is the objective value for solution i. As explained in the next sections, MOSA will be used at the method level of our bug localization approach in order to recommend the most relevant methods of the classes identified by the first step of our approach based on NSGA-II.

3.2.4 Fitness Functions

Both steps of our approach use two main fitness functions that are applied at the class level (global search) and the method level (local search). The first objective consists of the size of the solution which corresponds to the number of recommended classes or methods. The second objective of correctness is defined as the average of two functions: lexical-based similarity (LS) and history-based similarity (HS). Thus, we formally define this function as:

$$f_1 = \frac{LS + HS}{2} \tag{3.3}$$

The lexical-based similarity (LS) consists of an average of two functions. The first function is based on a cosine similarity *Tan et al.* (2006) between the description of a bug report and the source code while the second one checks the similarity between the description of a bug report and the API documentation. We used the whole content of a source code file (the code and comments). The vocabulary was extracted from the names of variables, classes, methods, parameters, types, etc. We used the Camel Case Splitter to perform the Tokenization for prepossessing the identifiers *Enslen et al.* (2009).

During the tokenization process, we used a standard information retrieval stop words to eliminate irrelevant information such as punctuation, numbers, etc. In addition, the words are reduced to their stem based on a Porter Stemmer. This operation reduces the deviation between related words such as "designing" and "designer" to the same stem "design". Then, the cosine similarity measure is used to compare between the description of a bug report and the source code (classes or methods).

Equation 3.4 calculates the cosine similarity between two vectors. Each actor is represented as an n dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the conceptual similarity between two vectors c_1 and c_2 is determined as follows:

$$\begin{aligned} Sim(c_1, c_2) = \cos(\overrightarrow{c_1}, \overrightarrow{c_2}) &= \frac{\overrightarrow{c_1}.\overrightarrow{c_2}}{\|\overrightarrow{c_1}\| \times \|\overrightarrow{c_2}\|} \\ &= \frac{\sum_{i=1}^{n}(w_{i,1} \times w_{i,2})}{\sqrt{\sum_{i=1}^{n}(w_{i,1})^2 \times \sum_{i=1}^{n}(w_{i,2})^2}} \in [0,1] \end{aligned} \tag{3.4}$$

where $\overrightarrow{c_1} = (w_{1,1}, ..., w_{n,1})$ is the term vector corresponding to actor c_1 and $\overrightarrow{c_2} = (w_{1,2}, ..., w_{n,2})$ is the term vector corresponding to c_2. The weights $w_{i,j}$ is computed using information retrieval based techniques such as the Term Frequency - Inverse Term Frequency (TF-IDF) method.

The second component of the correctness objective is the history-based similarity. This measure is an average of three functions. The first function counts the number of times that a code snippet (i.e. classes or methods) was fixed to eliminate bugs based on the history of bug reports. In fact, a source code that was fixed several times has a high probability of being buggy and includes new bugs. Formally, this function will be normalized between [0,1] and defined as:

$$H_1 = \frac{\sum_{i=1}^{Size(S)} NbFixedBugs(report, C_i)}{Size(S) \times Max(NbFixedBugs(report, C_i))} \tag{3.5}$$

where S is a solution containing a number of recommended classes
$S = \{c_1, c_2, \ldots, c_{size(S)}\}$. The second function checks if a code snippet (i.e. classes or methods) was recently changed or fixed. In fact, a source code that was modified recently has a higher probability of containing a bug. Thus, this function compares between the date of the bug report and the last date where the source code was modified. If a suggested code snippet was modified on the same day of the bug report then the value of this function is 1. We define this normalized function, normalized

in the range of [0,1] as following:

$$H_2 = \frac{\sum_{i=1}^{Size(S)} \frac{1}{report.data - last(report, c_i) + 1}}{Size(S)} \tag{3.6}$$

The third function evaluates the consistency between the recommended source code based on previous bug reports. The code snippets that are recommended together for similar previous bug reports have a high probability to include a bug evolving most of them. To this end, this function calculates the cardinality, Cbr, of the largest intersection set of code snippets (i.e. classes or methods) between the solution S and the sets of code snippets (i.e. classes or methods) recommended for each of previous bug reports. Then, this measure is normalized between [0,1] and defined as follows:

$$H_3 = \frac{Cbr}{Size(S)} \tag{3.7}$$

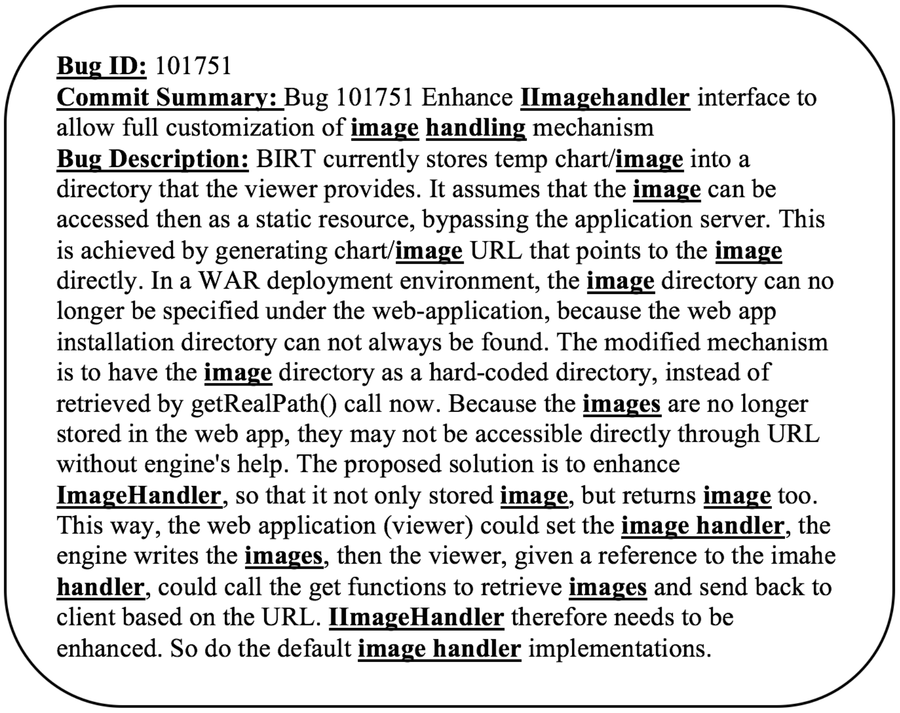
3.2.5 Class-Level Solution Approach

3.2.5.1 Solution Representation

To represent a candidate solution (individual), we used a vector representation. Each dimension of the vector represents a class to recommend for a specific bug report. Thus, a solution is defined as a sequence of classes to recommend for inspection by the developer to locate the bug.

When created, the order of recommended classes corresponds to their positions in the vector. The classes to recommend are dependent since a bug can be located in different classes. In addition, the goal is to recommend a minimum set of classes while maximizing the correctness objective.

For instance, A solution to find possible relevant classes for the bug report of Figure 3.3, extracted from our experiments, that shows an example of a bug report



Bug ID: 101751
Commit Summary: Bug 101751 Enhance **IImagehandler** interface to allow full customization of **image** **handling** mechanism
Bug Description: BIRT currently stores temp chart/**image** into a directory that the viewer provides. It assumes that the **image** can be accessed then as a static resource, bypassing the application server. This is achieved by generating chart/**image** URL that points to the **image** directly. In a WAR deployment environment, the **image** directory can no longer be specified under the web-application, because the web app installation directory can not always be found. The modified mechanism is to have the **image** directory as a hard-coded directory, instead of retrieved by getRealPath() call now. Because the **images** are no longer stored in the web app, they may not be accessible directly through URL without engine's help. The proposed solution is to enhance **ImageHandler**, so that it not only stored **image**, but returns **image** too. This way, the web application (viewer) could set the **image handler**, the engine writes the **images**, then the viewer, given a reference to the imahe **handler**, could call the get functions to retrieve **images** and send back to client based on the URL. **IImageHandler** therefore needs to be enhanced. So do the default **image handler** implementations.

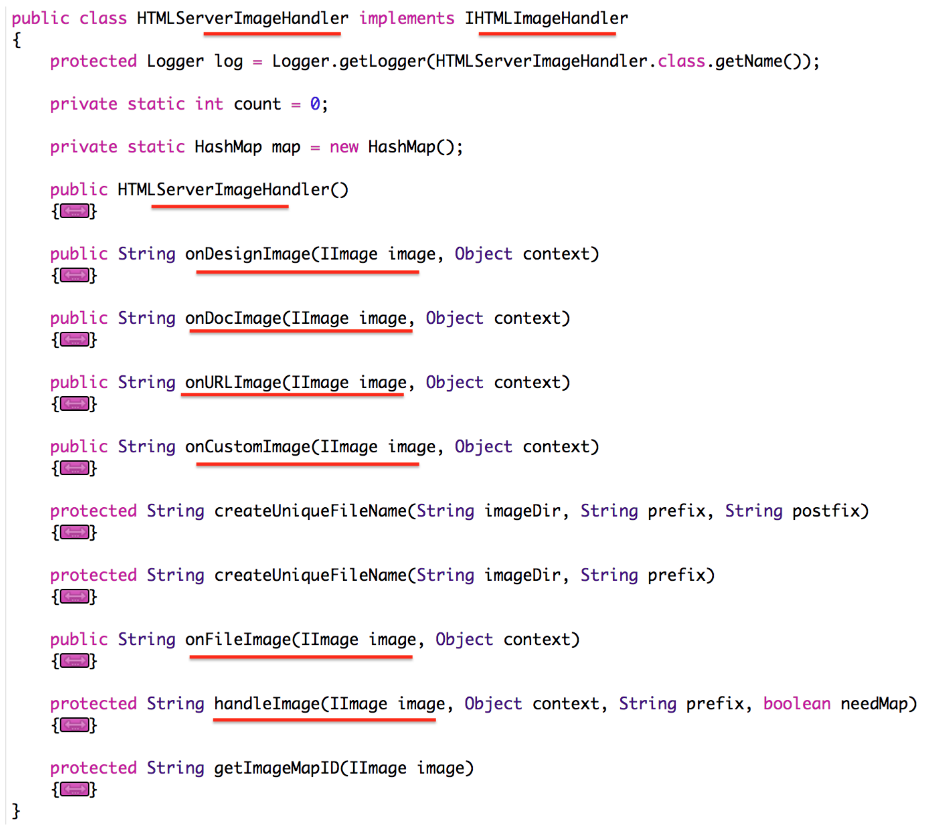
Figure 3.3: BIRT Bug Report Example (ID 101751)

from BIRT project (ID 101751) is a vector of several ranked classes to be inspected. This bug report describes a defect in the image handling mechanism. The solution consists of a sequence of classes to inspect extracted from the BIRT project.

3.2.5.2 Fitness Functions

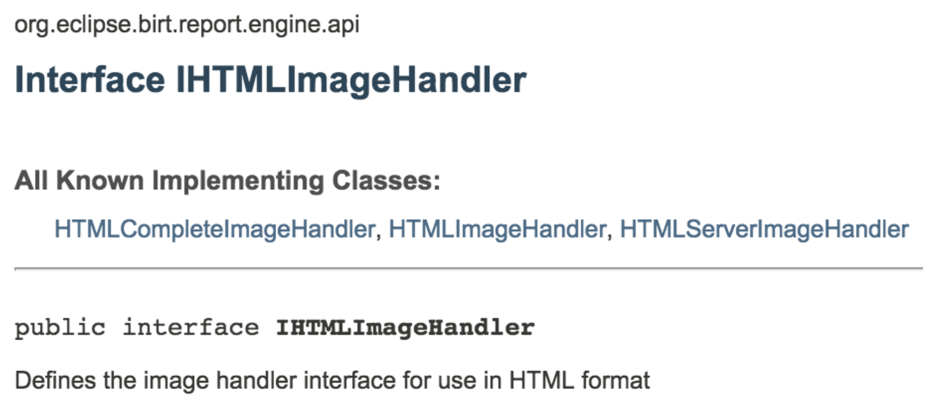
The first lexical similarity function is defined as the sum of the cosine similarity scores between a description of a bug report and the source code of each of the suggested classes divided by the total number of recommended classes. As described in Figures 3.4 and 3.5, the description of the bug report example includes several similar words with one of the recommended classes to inspect, the class *HTMLServerImageHandler*. Thus, the cosine similarity function applied between the source code of that class and the description of the bug report will detect such similarities. However, using only this similarity function may not be enough.

The text of a bug report is expressed in a natural language; however, a large



```
public class HTMLServerImageHandler implements IHTMLImageHandler
{
    protected Logger log = Logger.getLogger(HTMLServerImageHandler.class.getName());

    private static int count = 0;

    private static HashMap map = new HashMap();

    public HTMLServerImageHandler()
    {}

    public String onDesignImage(IImage image, Object context)
    {}

    public String onDocImage(IImage image, Object context)
    {}

    public String onURLImage(IImage image, Object context)
    {}

    public String onCustomImage(IImage image, Object context)
    {}

    protected String createUniqueFileName(String imageDir, String prefix, String postfix)
    {}

    protected String createUniqueFileName(String imageDir, String prefix)
    {}

    public String onFileImage(IImage image, Object context)
    {}

    protected String handleImage(IImage image, Object context, String prefix, boolean needMap)
    {}

    protected String getImageMapID(IImage image)
    {}
}
```

Figure 3.4: A code fragment from the class HTMLServerImageHandler


org.eclipse.birt.report.engine.api

Interface IHTMLImageHandler

All Known Implementing Classes:

HTMLCompleteImageHandler, HTMLImageHandler, HTMLServerImageHandler

```
public interface IHTMLImageHandler
```

Defines the image handler interface for use in HTML format

Figure 3.5: API Specification of the interface IHTMLImageHandler

part of the content of source code is described in a programming language (except comments). Thus, the similarity score between a bug report description and a source code will be higher in case of extensive use of comments in the code or if the bug report clearly uses the names of code elements or code snippet (which is not always the case in bug reports). Thus, we added the new similarity measure between the bug reports and the APIs description to deal with situations where there is no enough comments in the code or no code snippet in the bug report.

The second lexical function is based on the use of cosine similarity between the bug report description and the API specification of each method of a recommended buggy class. Thus, it is defined as the sum of the maximum of the cosine similarity scores between a description of a bug report and each of the methods composing the suggested class divided by the total number of recommended classes. Figure 3.5 shows the API specification of the *IHTMLImageHandler* interface that includes different terms such as image and handler that also exists in the bug report description of Figure 3.4. Thus, the lexical similarity between the API specification and the description of a bug report may also help to better identify relevant buggy classes.

In addition to lexical functions, we also add another component to represent the historical measure which composes of three functions. The first function counts the number of times a particular class was fixed. Normally, the more times developers

Bug#	**Commit Description**	**Date**
Bug 243553	HTMLServerImageHander returns wrong ImageUrl when using HTMLRenderOption	Aug 2008
Bug 101751	Enhance IImagehandler interface to allow full customization of birt image handling mechanism	July 2005
Bug 200187	Deprecated HTMLServerImageHandler methods are not marked as deprecated	July 2007

Table 3.1: List of commits reported on the same file (HTMLServerImageHander) for Birt Project

make changes in a class, the more defects could be introduced in this particular file in the future. The second history-based function is to measure how recent a particular class has been changed because making some changes today might cause some defects to happen tomorrow. The last function in this category is to evaluate the consistency between what we recommend in terms of classes to what has been touched by the developers in the past to fix a particular defect. Table 3.1 shows a list of commits reported on the same file in Figure 3.4, we use the history of bug reports to find the similarity in the description of several commits/bug reports and therefore find the classes or methods that were fixed in the past in order to build our solution for the current reported bug.

3.2.6 Method-Level Solution Approach

3.2.6.1 Solution Representation

We used a vector of elements to represent a candidate solution, each element represents a method along with its class name in order to recommend for a given bug. Thus, a solution is defined as a sequence of methods to recommend for inspection by the developer to locate the bug. The recommended methods are sorted and ranked in their vector to represent their degree of importance to be reviewed by the developers.

onCustomImage	handleImage	onDesignImage

Figure 3.6: A simplified method-level solution representation

The methods to recommend are dependent since a bug can be located in different methods among different classes while maintaining the balance between minimizing the set of methods to recommend and maximizing the value of the correctness objective.

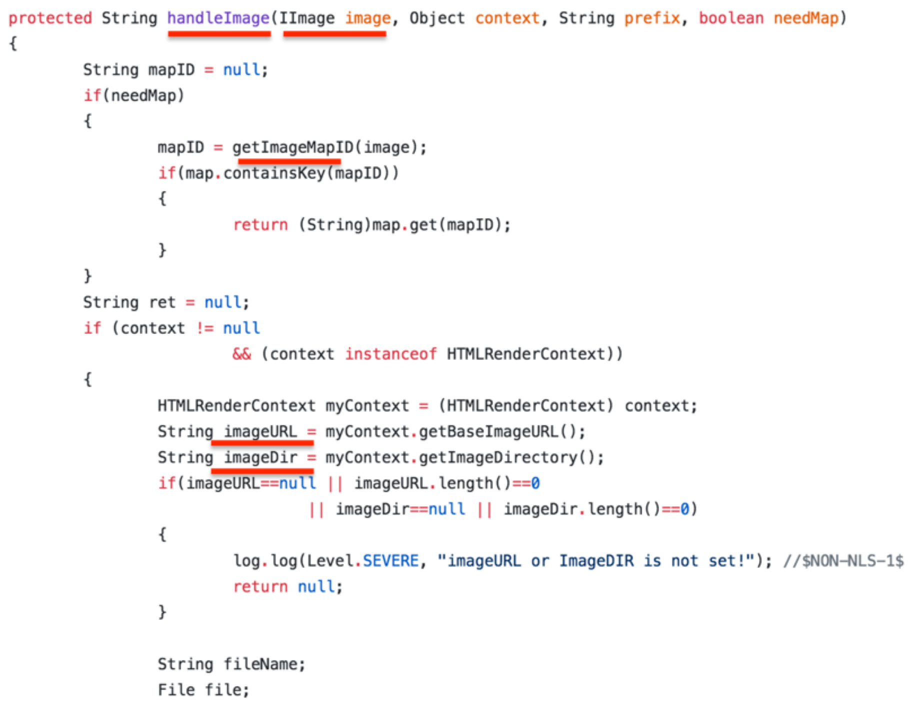
Figure 3.6 shows a simplified solution generated to find possible relevant methods for the bug report of Figure 3.3 extracted from the BIRT project (ID 101751).

3.2.6.2 Fitness Functions

We adapted the fitness functions defined at the class level to calculate the new method level measures. Lexical similarity functions are used to weigh the similarity between the source code of a suggested method from one side or the description of a bug report and the API specification of each method from the other side. As highlighted in Figure 3.7, the description of the bug report includes a few keywords that already exist in the *handleImage* method. Therefore, the similarity measure between the source code of that method and the description of the bug report is high. History-based fitness functions are used on a particular method by looking at its recently applied changes; along with the consistency between our recommended methods to previously fixed methods of similar bugs in the past.

3.3 Evaluation

In order to evaluate our approach for recommending relevant methods to inspect for bug reports, we conducted a set of experiments based on different versions of 6 open source systems listed in Table 3.2. Each experiment is repeated 30 times, and



```
protected String handleImage(IImage image, Object context, String prefix, boolean needMap)
{
        String mapID = null;
        if(needMap)
        {
                mapID = getImageMapID(image);
                if(map.containsKey(mapID))
                {
                        return (String)map.get(mapID);
                }
        }
        String ret = null;
        if (context != null
                        && (context instanceof HTMLRenderContext))
        {
                HTMLRenderContext myContext = (HTMLRenderContext) context;
                String imageURL = myContext.getBaseImageURL();
                String imageDir = myContext.getImageDirectory();
                if(imageURL==null || imageURL.length()==0
                                || imageDir==null || imageDir.length()==0)
                {
                        log.log(Level.SEVERE, "imageURL or ImageDIR is not set!"); //$NON-NLS-1$
                        return null;
                }

                String fileName;
                File file;
```

Figure 3.7: A code fragment from the method handleImage

the obtained results are subsequently statistically analyzed. Our aim is to compare our hybrid multi-objective approach with a variety of existing approaches:

- Approaches not based on heuristic search such as *Nguyen et al.* (2011); *Ye et al.* (2016a); *Zhou et al.* (2012); *Ye et al.* (2016a) at the class and methods level.
- Our previous multi-objective work *Almhana et al.* (2016), a one step multi-objective formulation based on NSGA-II to identify relevant classes
- A mono-objective formulation.

In this section, we present our research questions followed by experimental settings and parameters. Finally, we discuss our results for each of those research questions.

3.3.1 Research Questions

In our study, we assess the performance of our approach by finding out whether it could identify the most relevant classes and methods to inspect for bug reports.

Project	# bugs	Time	# API	# files in the project (average per version)	# methods in the project (median)	# fixed files/-classes per bug report (median)	# fixed methods per bug report (median)
Eclipse UI	6495	10/2001 01/2014	1314	3454	29582	2	2
Birt	4178	06/2005-12/2013	957	6841	57329	1	3
JDT	6274	10/2001 01/2014	1329	8184	30240	2	2
AspectJ	593	03/2002-01/2014	54	4439	21346	2	2
Tomcat	1056	07/2002 01/2014	389	1552	17970	1	2
SWT	4151	02/2002-01/2014	161	2056	28355	3	5

Table 3.2: Studied Projects

Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. The main question to answer is to what extent the proposed approach can propose meaningful bug localization solutions based on the description of a bug report? To this end, we defined the following research questions:

- **RQ1.** (Effectiveness) To what extent can the proposed approach identify relevant methods to localize bugs based on bug reports description?
- **RQ2.** (Comparison to search techniques) How does the proposed hybrid approach performs comparing to our previous multi-objective work *Almhana et al.* (2016), a one step multi-objective formulation based on NSGA-II to identify relevant methods, random search, and a mono-objective formulation?
- **RQ3.** (Comparison to state-of-the-art) How does our approach perform compared to existing bugs localization techniques not based on heuristic search?

To answer RQ1, we validate the proposed multi-objective technique on six medium to large-size open-source systems, as detailed in the next section, to evaluate the correctness of the recommended methods to inspect for a bug report. To this end, we used the following evaluation metrics:

- **Precision@k** is the fraction of two components. The numerator component is the number of correct recommended methods in the top k of recommended methods (or files) in the solution. The denominator component is the minimum number of methods (or files), between k and the number of recommended methods/files to inspect in the ranked recommendations list.
- **Recall@k** is the fraction of two components. The numerator component is the number of correct recommended methods in the top k of recommended methods (or files) in the solution. The denominator component is the total number of expected methods (or files) to recommend that contain the bug.

- **Accuracy@k** measures the percentage of bug reports for which at least one correct recommendation was provided in the top k ranked methods (or files).

To answer RQ2, we compared, using the above metrics, the performance of our hybrid multi-objective approach, called Hybrid Multi-objective Approach (HMOA), with our previous multi-objective work *Almhana et al.* (2016), a one step multi-objective formulation based on NSGA-II to identify relevant methods (called One Step Multi-objective Formulation based on NSGA-II (hNSGA-II)), random search and a mono-objective formulation, based on a Genetic Algorithm, aggregating all the objectives into one objective with equal weight. We note that hNSGA-II is using only NSGA-II using the same fitness functions but applied directly at the methods level (minimizing the number of recommended methods and maximizing the relevance of recommended methods). Furthermore, we implemented three mono-objective formulations: 1.with an equal aggregation of both objectives (Genetic Algorithm (GA)); 2. a mono-objective algorithm with the only objective of lexical similarity (Lexical Similarity (LS)); and 3. a mono-objective algorithm with the only objective of history similarity (History Similarity (HS)). Random search and the mono-objective formulation are applied for both levels (class and method levels) similar to our approach so we can ensure a fair comparison. hNSGA-II is used to show the value of using a two levels approach.

If Random Search outperforms a guided search method thus, we can conclude that our problem formulation is not adequate. It is important also to determine if our objectives are conflicting and outperform a mono-objective technique. The comparison between a multi-objective technique with mono-objective ones is not straightforward. The first one returns a set of non-dominated solutions while the second one returns a single optimal solution. To this end, we choose the nearest solution to the Knee point *Deb et al.* (2002) (i.e., the vector composed of the best objective values among the population members) as a candidate solution to be compared with the single solution returned by the mono-objective algorithm. We did not invent the knee point method and we used it as recommended by the current literature *Keller* (2019); *Emmerich and Deutz* (2018); *Deb and Gupta* (2011). The two common ways are the use of the reference point and the knee point. The definition of the reference point (best region of the Pareto front) can be subjective for most real-world problems since it depends on the preferences *Keller* (2019). The knee point represents the maximum trade-off between the objectives thus it is reasonable to compare it with a mono-objective solution with equal weights of the different objectives aggregated in one fitness function. The fact that we are comparing a mono-objective formulation with equal weights to a knee point (representing the maximum possible trade-off) ensures a fair comparison.

The hNSGA-II algorithm identifies relevant methods for bug reports using the same fitness functions, applied at the methods level, of the proposed approach but only using one step. Thus, the solutions of hNSGA-II are a sequence of methods that are generated and evolved using NSGA-II. The comparison with hNSGA-II is important to confirm the relevance of using a hybrid approach combining both a global and local search algorithms. In comparison with our previous class level work *Almhana et al.* (2016), we considered the files of the methods to ensure a fair comparison. This comparison can evaluate the impact of adding the MOSA component on the quality of the results especially in terms of finding the best ranking of the classes/files.

To answer RQ3, we compared our multi-objective approach to different existing techniques not based on heuristic search:

- BugScout *Nguyen et al.* (2011) identifies relevant classes based on the use of Latent Dirichlet Allocation measure *Blei et al.* (2003).
- BugLocator *Zhou et al.* (2012) ranks classes using both textual and structural similarity.
- Learning-to-rank (LRank) *Ye et al.* (2016a) technique ranks methods using a

machine learning technique to learn from the history of previous bug reports. While this technique can be adopted to both class and methods level like our approach, we configured the implementation to recommend methods as output. Also, we compared our work with two additional baselines. The first one is only based on the use of the lexical measure (LS) to rank classes and the second one is based on the only use of the history measure (HS). These two baselines may justify or not the need of considering complementary information from both the lexical and history similarities in our multi-objective formulation.

We considered the files of the methods to ensure a fair comparison with class level recommendation tools (BugScout, BugLocator and our previous work *Almhana et al.* (2016)) and we considered comparisons of the results at the methods level with *Ye et al.* (2016a). Thus, we have two categories of comparison: 1) the class-level approaches are compared to our approach using the evaluation metrics applied at the files (precision, recall and accuracy); and 2) the methods-level approaches are compared to our approach using the evaluation metrics applied at the methods (precision, recall and accuracy).

3.3.2 Software Projects and Experimental Setting

As described in Table 3.2, we extended a benchmark data sets for six open-source systems *Ye et al.* (2014, 2016a) from the class to the methods level and we are making this new benchmark available to the community *Almhana and Kessentini* (2020b). The data is a spreadsheet where rows are bugs and columns are attributes of the bug. Columns are bug id, bug description, bug summary/commit, bug commit id, bug resolved date. Besides, we added two more columns, the first column contains a list of classes that have been changed in order to resolve the bug, and the second contains a list of methods that have been fixed. Both of those columns (classes list and method list) have been generated using Git (version code system) which provides us the ability to extract a list of files or methods that have been changed in each commit along with date and developer (committer). The whole data has been generated using data from Git or Bugzilla (bug tracking system).

- **Eclipse UI** is the user interface of the Eclipse development framework.
- **Tomcat** implements several Java EE specifications.
- **AspectJ** is an aspect-oriented programming (AOP) extension created for the Java programming language.
- **Birt** provides reporting and business intelligence capabilities.
- **SWT** is a graphical widget toolkit.
- **JDT** provides a set of tool plug-ins for Eclipse.

In Table 3.2 shows the different statistics of the analyzed systems including the time range of the bug reports, the number of bug reports, the number of classes and methods in a project, the number of APIs, the number of fixed classes per bug report, and the number of fixed methods per bug report. The total number of collected bug reports and associated classes and methods is more than 22,000 bug reports for the six open source systems. All these projects are using BugZilla tracking system and GIT as a version control system. The ground truth used in our evaluation is the bug location and its respective bug report. To avoid using a fixed version of the source code, we associated a before-fixed version of the source code to each bug report. Therefore, for each bug report in our evaluation, we used the version of the source code just before the fix was committed. Based on the collected data, we created two sets: one for the training data and the other for the test data. The bug reports for each system were sorted chronologically based on the time dimension. The sorted bug reports are then split into 10 folds with equal sizes, where fold$_1$ contains the most oldest bug reports and the last fold, fold$_{10}$, contains the recent ones. In addition, the oldest fold is split into 70% training (history of bug reports) and 30% validation. The approach is trained on fold $i+1$ and tested on fold$_i$, for all i from 1 to 10. The best recommended solution is then compared with the expected solution of classes and methods that contain the bug. Thus, fold$_1$ contains the oldest bug reports whereas fold$_{10}$ contains the latest bug reports. Since the folds are arranged chronologically, this means that the system is always trained on the most recent bug reports with respect to the testing fold.

3.3.3 Parameters Tuning and Statistical Tests

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another.

We used the Wilcoxon rank sum test *Wilcoxon et al.* (1970) in a pairwise fashion in order to detect significant performance differences between the algorithms (HMOA vs each of the competitors) under comparison based on 30 independent runs. BugScout and BugLocator are both deterministic thus we did not perform 30 independent runs.The Wilcoxon test allows testing the null hypothesis H0 that states that both algorithms medians' values for a particular metric are not statistically different against H1 which states the opposite. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values' ranks instead of operating on the values themselves. Since we are comparing more than two different algorithms, we performed several pairwise comparisons based on Wilcoxon test to detect the statistical difference in terms of performance. To compare two algorithms based on a particular metric, we record the obtained metric's values for both algorithms over 30 runs. For deterministic techniques, we considered one value of each metric on each system. After that, we compute the metric's median value for each algorithm. Besides, we executed the Wilcoxon test with a 95% confidence level ($\alpha = 5\%$) on the recorded metric's values using the Wilcoxon MATLAB routine. If the returned p-value is less than 0.05 then we reject H0 and we can state that one algorithm outperforms the other, otherwise we cannot say anything in terms of performance difference between the two algorithms.

The Wilcoxon test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. To this end, we used the Vargha and Delaney's A statistics which is a non-parametric effect size measure. In our context, given the different performance metrics (such as Precision and Recall), the A statistics measures the probability that running an algorithm B1 (HMOA) yields better performance than running another algorithm B2 (such as GA). If the two algorithms are equivalent, then A = 0.5.

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. In fact, the parameter setting significantly influences the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and each system, we performed a set of experiments using several population sizes: 10, 20, 30, 40 and 50. The stopping criterion was set to 100,000 fitness evaluations for all search algorithms to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion since our approach requires multiple objectives. Each algorithm was executed 30 times with each configuration and then the comparison between the configurations was performed based on different metrics described previously using the Wilcoxon test. The other parameters values were fixed by trial and error and are as follows: (1) crossover probability = 0.4; mutation probability = 0.3 where the probability of gene modification is 0.1. In fact, we decided to reduce the diversity of the generated solutions at each iteration since a local search exploration will be executed as well as a second step.

MOSA was performed with a starting temperature of 0.0002 and an alpha value of 0.99995. The starting temperature and alpha values were chosen because they yielded the best results in empirical preliminary tests. All probability distributions used by the search process (e.g., to determine the type of mutation to execute or code fragments to select) were such that each discrete possibility had an equal chance of being selected.

3.3.4 Results

3.3.4.1 Results for RQ1

The results of Table 3.3 and Figures 3.8-3.13 confirm the effectiveness of our hybrid multi-objective approach (HMOA) to identify the most relevant classes and methods for bug reports that include the bugs on the 6 open source systems. Table 3.3 shows the average precision@k results of our HMOA technique on the different six systems, with k ranging from 5 to 20. For example, most of the recommended methods to inspect in the top 5 (k=5) are relevant with a precision of 83%. The lowest precision is around 71% for k=20 which still could be considered acceptable due to the low granularity/abstraction level (methods). In terms of recall, Table 3.3 confirms that the majority of the expected methods to recommend are located in the top 20 (k=20) with an average recall score of 83%. An average of more than 73% of methods recommended in the top 5 covers the expected buggy methods. The average accuracy@k results on the different six systems are described in Table 3.3 showing that an average of 67%, 74%, 86%, and 91% are achieved for k = 5, 10, 15, and 20 respectively.

Figures 3.11-3.13 summarize the results of the precision@10, recall@10 and accuracy@10 for each of the studied systems. The obtained results clearly show that most of the buggy methods were recommended correctly by our hybrid multi-objective approach in the top 10 with a minimum precision of 82% for AspectJ, a minimum recall of 84% for Eclipse and a minimum accuracy of 81% for Eclipse as well. Thus, we noticed that our technique does not have a bias towards the evaluated system. As

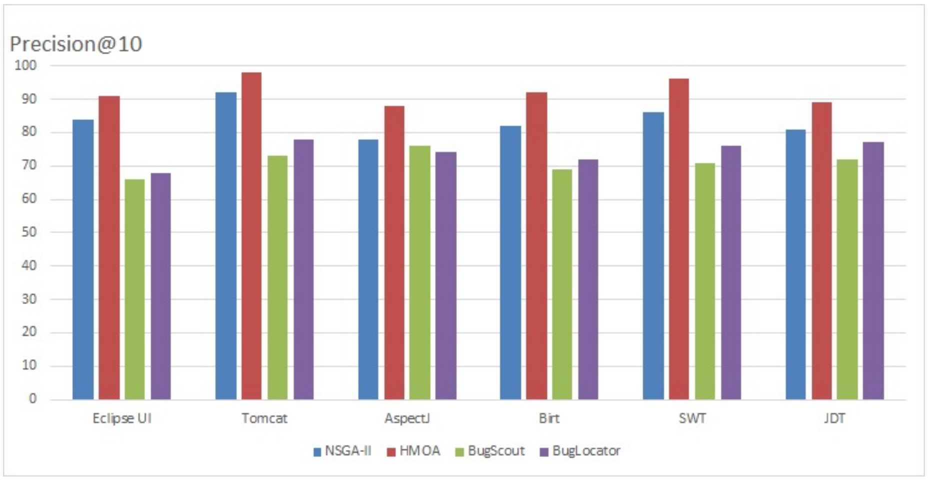


Figure 3.8: Median Precision@10 at the class level on the different systems for 30 independent runs.

described in Figures 3.11-3.13, in all systems, we had almost similar average scores of precision, recall, and accuracy. All these results based on the different measures were statistically significant on 30 independent runs using the Wilcoxon test with a 95% confidence level ($\alpha < 5\%$) as detailed in Table 3.5.

To answer RQ1, the obtained results on the six open source systems using the different evaluation metrics of precision, recall, and accuracy clearly validate the hypotheses that our hybrid multi-objective approach can recommend efficiently relevant buggy methods to inspect for each bug report.

3.3.4.2 Results for RQ2

Concerning RQ2, we have two categories of comparison. The first category is dedicated to the comparison of HMOA with other method level approaches (LS, LRank, HS, GA, RS, hNSGA-II) to our approach. The second category is related to the comparison of HMOA with our previous multi-objective work *Almhana et al.* (2016) for bugs localization at the class level. Thus, the comparison in the second category is performed at the class level (similar to RQ3).

Tables 3.3-3.4 and Figures 3.8-3.10 confirm that HMOA is better, in average, than

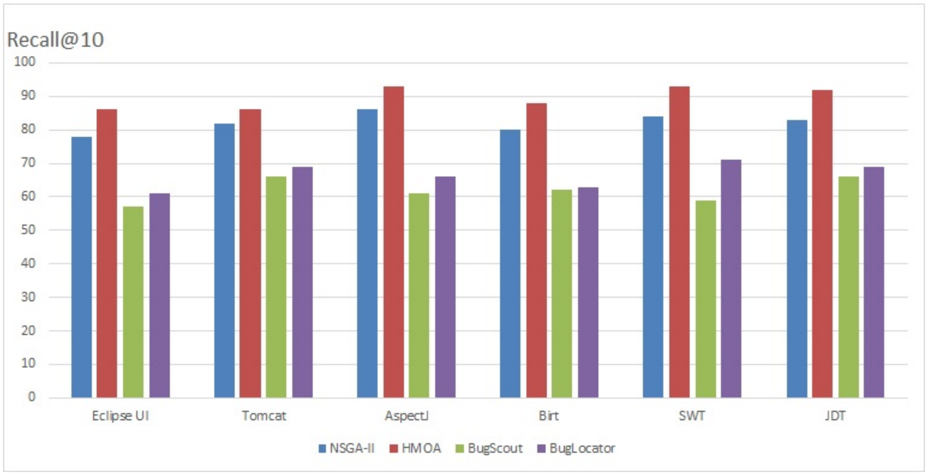


Figure 3.9: Median Recall@10 at the class level on the different systems for 30 independent runs.

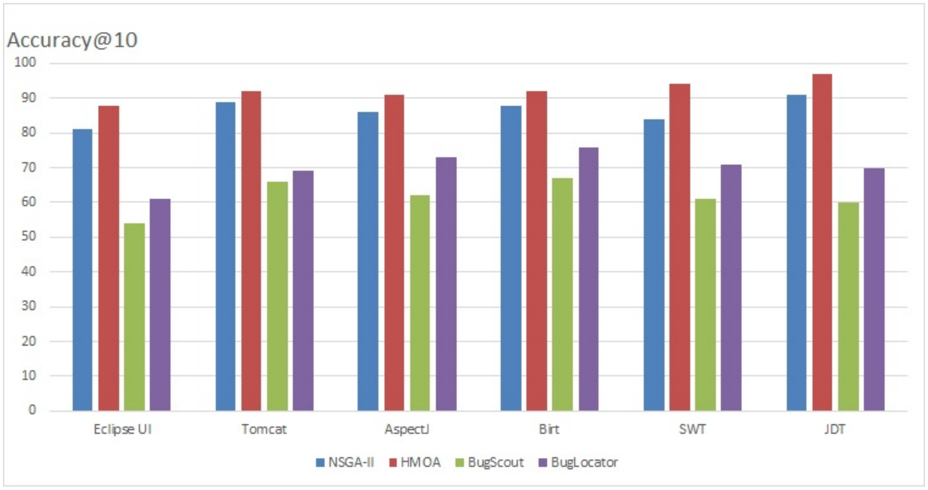


Figure 3.10: Median Accuracy@10 at the class level on the different systems for 30 independent runs.

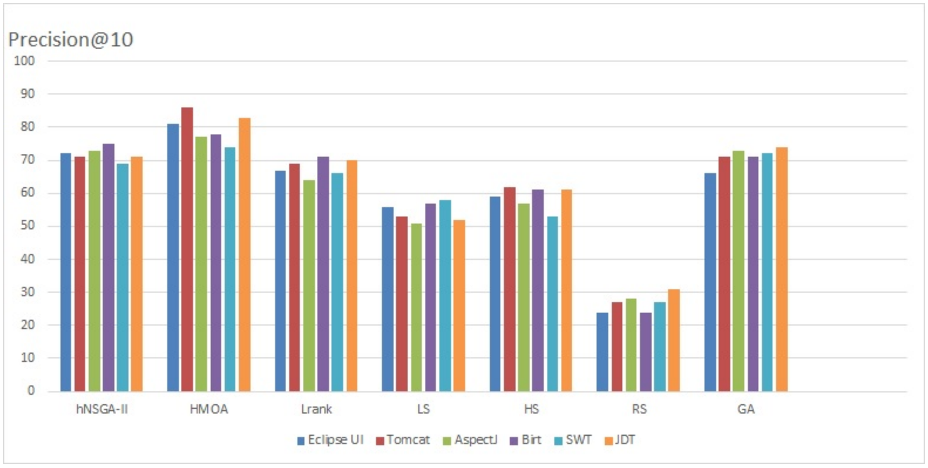


Figure 3.11: Median Precision@10 at the methods level on the different systems for 30 independent runs.

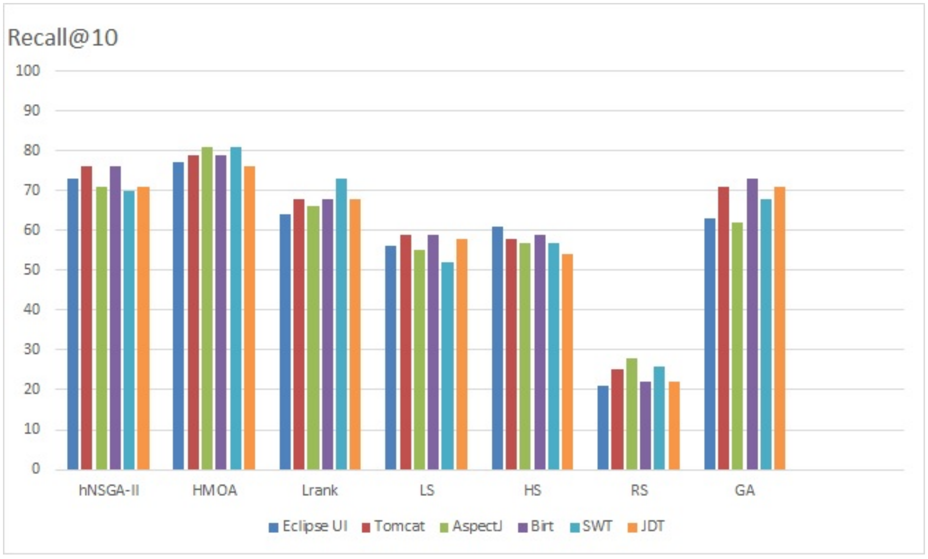


Figure 3.12: Median Recall@10 at the methods level on the different systems for 30 independent runs.

K	**Precision @ K**						
	hNSGA-II	**HMOA**	LR	LS	HS	RS	GA
5	76	**83**	72	62	66	32	69
10	71	**79**	68	54	58	26	71
15	68	**76**	61	51	54	28	63
20	64	**71**	52	44	49	21	54
K	**Recall @ K**						
	hNSGA-II	**HMOA**	LR	LS	HS	RS	GA
	69	**73**	61	49	51	21	58
	72	**78**	67	54	56	24	63
	75	**81**	69	59	62	27	71
	79	**83**	72	63	66	21	74
K	**Accuracy @ K**						
	hNSGA-II	**HMOA**	LR	LS	HS	RS	GA
5	64	**67**	58	39	34	23	51
10	69	**74**	64	52	48	27	57
15	81	**86**	77	61	57	29	63
20	86	**91**	83	68	66	33	72

Table 3.3: Median Precision@k, Recall@k and Accuracy@k on 30 independent runs at the methods level.

K	**Precision @ K**			
	NSGA-II	**HMOA**	Bug Scout	Bug Locator
5	89	**100**	76	78
10	82	**92**	71	74
15	74	**84**	63	69
20	68	**81**	48	51
K	**Recall @ K**			
	NSGA-II	**HMOA**	Bug Scout	Bug Locator
	72	**84**	59	62
	81	**86**	64	67
	87	**89**	69	72
	94	**100**	74	80
K	**Accuracy @ K**			
	NSGA-II	**HMOA**	Bug Scout	Bug Locator
5	68	**83**	41	44
10	86	**86**	62	69
15	94	**97**	74	78
20	97	**100**	79	82

Table 3.4: Median Precision@k, Recall@k and Accuracy@k on 30 independent runs at the class/files level.

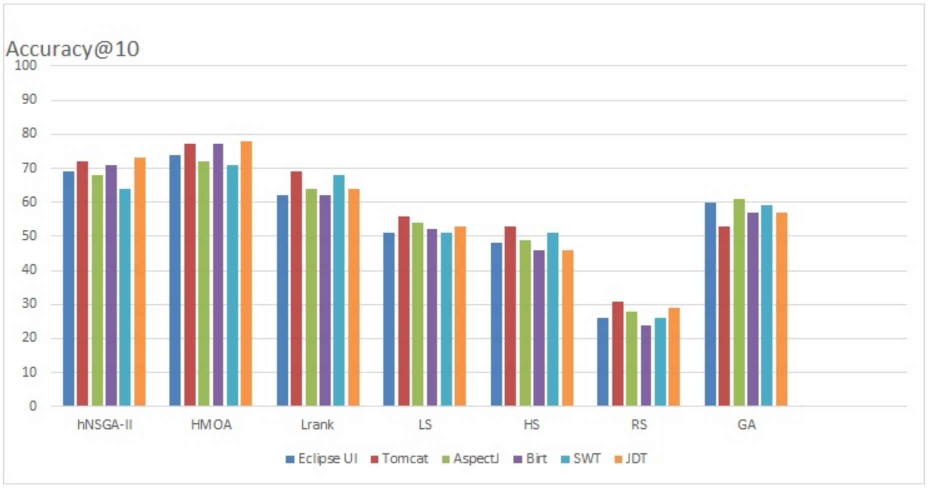


Figure 3.13: Median Accuracy@10 at the methods level on the different systems for 30 independent runs.

random search, the one-step multi-objective methods level formulation (hNSGA-II), and the three mono-objective formulations (LS, HS and GA) based on the three metrics of precision, recall and accuracy on all the 6 systems.

The average accuracy, precision, and recall values of random search (RS) on the six systems are lower than 32% as described in Table 3.3. This can be explained by the huge search space to explore to identify the best order of methods to inspect for bugs localization. The performance of the three mono-objective algorithms was much better than random search but lower than the multi-objective formulations. The aggregation of both objectives into one objective generates better results on all the six systems than the two other algorithms considering each objective separately. Thus, an interesting observation is the clear complementary between the history-based similarity function and the lexical-based measure. In fact, we found that the buggy methods that are not detected by one of the two algorithms were identified by the other algorithm. The average precision, recall, and accuracy of each of the two algorithms (LH and HS) was between 61% and 73% but the aggregation of both objectives into one in our multi-objective formulations improve a lot the obtained results. In addition, since the three multi-objective formulations (NSGA-II, MOHA, and hNSGA-II) outperform the mono-objective GA then it is clear that the two objectives of correctness/relevance and the number of recommended methods are conflicting.

Table 3.4 confirms also the outperformance of our hybrid multi-objective algorithm comparing to the remaining multi-objective formulations (hNSGA-II and NSGA-II). It is clear that HMOA results are better than hNSGA-II in terms of precision, recall and accuracy. This may confirm that the use of MOSA as a local search to identify methods helped for a better exploration of the large space of possible method comparing to the one-step NSGA-II approach. Furthermore, the results of Figures 3.8-3.10 show that both HMOA have better precision, recall and accuracy, on average, than previous work *Almhana et al.* (2016). Thus, it is also clear that adaptation of the methods level fitness functions is more adequate than our previous work to localize bugs and their impact on the ranking of the classes to be explored by the developers in a positive way.

All these results were statistically significant on 30 independent runs using the Wilcoxon test with a 95% confidence level ($\alpha < 5\%$) as described in Table 3.5. We have also found the following results of the Vargha Delaney A_{12} statistic : a) On large and medium scale systems (Birt, JDT, Eclipse UI, and AspectJ) HMOA is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.89; b) On small scale systems (Tomcat, SWT), HMOA is better than all the other algorithms with an A effect size higher than 0.91.

We conclude that there is empirical evidence that our hybrid multi-objective formulation surpasses the performance of random search and other search-based approaches thus our formulation is adequate (this answers RQ2).

3.3.4.3 Results for RQ3

Since it is not sufficient to compare our approach with only search-based algorithms, we compared the performance of NSGA-II with three different bug localization techniques not based on heuristic search *Nguyen et al.* (2011); *Ye et al.* (2016a); *Zhou et al.* (2012). Similar to the comparison with NSGA-II, we used class-level comparison measures for *Nguyen et al.* (2011); *Zhou et al.* (2012) and method-level comparison for *Ye et al.* (2016a). Tables 3.3 and 3.4, and Figures 3.8-3.13 present the precision@k, recall@k and accuracy@k results for the 3 implemented methods, with k ranging from 5 to 20. HMOA achieves better results, on average, than the other three methods on all six projects. For example, our approach achieved, on average, Precision@k of 92%, 87%, 79% and 76% are achieved for k= 5, 10, 15 and 20 respectively as described in Table 3.4. In comparison, BugLocator achieved an average Precision@k of 68%. BugScout and Lrank achieved an average Precision@k of 66% and 72%, respectively. Similar observations are also valid for the recall@k and accuracy@k.

Based on the results of Figures 3.11-3.13 Birt and Tomcat are two projects where Lrank performs close to the HMOA approach. For many bug reports in Birt, most of the buggy methods are those that have been frequently fixed in previous bug reports which explain the relatively high performance obtained by Lrank and HMOA. Since the bug fixing information is exploited by both the NSGA-II approach and Lrank, it is expected that they obtain the best performance results.

To answer RQ3, the obtained results on the six open source systems using the different evaluation metrics of precision, recall and accuracy clearly validate the hypotheses that our hybrid multi-objective approach outperforms several bugs localization techniques not based on heuristic search both at the method and class levels.

Precision	**hNSGA-II**	**NSGA-II**	**BugScout**	**BugLocator**	**Lrank**	**LS**	**HS**	**RS**	**GA**
Eclipse UI	0.012	0.024	0.014	0.021	0.032	0.023	0.001	0.003	0.027
Tomcat	0.038	0.013	0.017	0.011	0.017	0.017	0.013	0.012	0.014
AspectJ	0.022	0.024	0.021	0.017	0.037	0.021	0.004	0.017	0.011
Birt	0.016	0.047	0.018	0.003	0.032	0.031	0.012	0.031	0.023
SWT	0.038	0.014	0.022	0.014	0.024	0.011	0.024	0.004	0.014
JDT	0.021	0.035	0.017	0.019	0.017	0.023	0.012	0.014	0.027
Recall	**hNSGA-II**	**NSGA-II**	**BugScout**	**BugLocator**	**Lrank**	**LS**	**HS**	**RS**	**GA**
Eclipse UI	0.023	0.020	0.027	0.023	0.027	0.026	0.017	0.002	0.031
Tomcat	0.031	0.017	0.004	0.007	0.032	0.011	0.031	0.012	0.023
AspectJ	0.014	0.019	0.016	0.016	0.018	0.007	0.014	0.006	0.014
Birt	0.022	0.014	0.011	0.012	0.019	0.024	0.022	0.011	0.017
SWT	0.031	0.023	0.016	0.032	0.031	0.016	0.016	0.013	0.023
JDT	0.023	0.011	0.021	0.037	0.043	0.018	0.027	0.014	0.019
Accuracy	**hNSGA-II**	**NSGA-II**	**BugScout**	**BugLocator**	**Lrank**	**LS**	**HS**	**RS**	**GA**
Eclipse UI	0.026	0.032	0.026	0.018	0.034	0.007	0.013	0.024	0.011
Tomcat	0.028	0.017	0.017	0.022	0.021	0.016	0.017	0.008	0.023
AspectJ	0.031	0.024	0.032	0.016	0.038	0.023	0.022	0.013	0.017
Birt	0.017	0.019	0.021	0.024	0.027	0.009	0.031	0.011	0.032
SWT	0.024	0.027	0.019	0.019	0.021	0.017	0.024	0.021	0.037
JDT	0.006	0.021	0.024	0.027	0.013	0.023	0.011	0.017	0.021

Table 3.5: The Wilcoxon rank sum test results in a pairwise fashion (HMOA vs each of the competitors) to detect significant performance differences between the algorithms under comparison using the Precision, Recall and Accuracy measures.

3.4 Discussion

We executed our hybrid multi-objective algorithm on a desktop computer with CPU Intel(R) Core(TM) i7 3.2 GHz and 20G RAM. Figure 3.14 presents the average execution time of our approach on 30 independent runs for the different six systems. This average execution time is to parse all bug reports for single system and generate the recommended solutions. We have also compared the HMOA execution time to our previous work based on NSGA-II to evaluate the cost of adding the new MOSA component to localize bugs at the method level. The average execution time on the different systems was around 23 minutes. The highest execution time was observed on the Eclipse system with 28 minutes and the lowest one was around 19 minutes for AspectJ. We believe that the execution is reasonable since bug localization is not a real-time problem. We also found that the execution time depends on the number of files to parse and the history of bug reports. Furthermore, the cost of adding the MOSA local search is low with an average of 6 minutes comparing to our previous work based on NSGA-II at the class level. Furthermore, we compared the execution time between our approach and hNSGA-II which shows that the local search based on MOSA is actually faster than applying NSGA-II for the methods-level search (an average of around 3 mins per system). In fact, the hNSGA-II formulation is executed at the methods level which is a much larger search space than the use of local search on a smaller search space of classes identified after a number of iterations of NSGA-II at the class level.

To evaluate the impact of increasing the size of the data used (history of previous bug reports and changes), we executed a scenario on the JDT project in which we increased the size of the dataset incrementally fold by fold until we include all the 9 folds in the dataset. It is clear from Figure 3.15 that for all the three metrics of Precision@k, Recall@k and Accuracy@k that increasing the size of the previous bug reports does not improve all the three metrics. This can be explained by the fact that

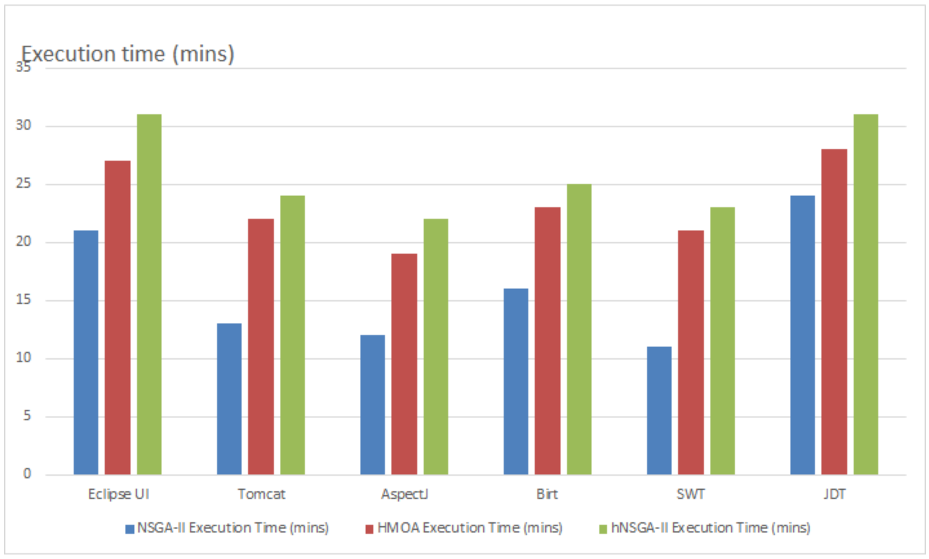


Figure 3.14: Average execution time (in minutes) of NSGA-II, hNSGA-II and HMOA, on the different systems for 30 independent runs on the different systems

recent bug reports and history of changes are the most important part of the data. The obtained results confirm also that our hybrid multi-objective approach did not require a large set of data to generate good results in terms of finding possible buggy methods for bug reports. One interesting observation from the recall results is that this measure did not decrease when more bugs reports are added to the datasets. It could be explained by the fact that the the history-based part of the fitness function is only part of the objective, thus the noise introduced by older bug reports is not very impactful. Futhermore, our approach is not based on machine learning to learn from all the dataset. It is based on metaheuristics search guided by fitness functions thus the results are likely less susceptible to noise.

3.5 Threats to Validity

We want to acknowledge several threats to the validity of the paper such as the factors that can bias our empirical study. These factors can be classified into three categories: internal validity, construct internal, and external validity. Construct va-

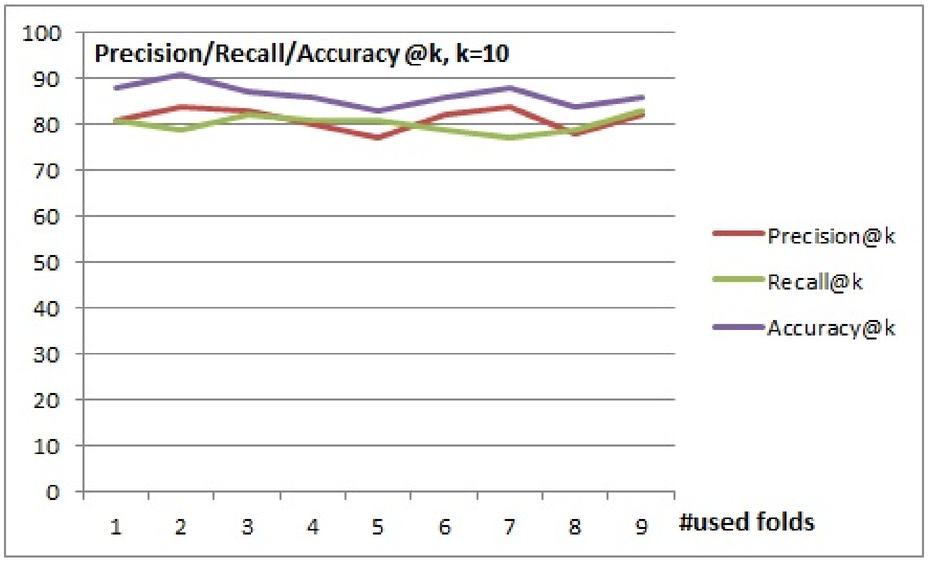


Figure 3.15: Impact of the data training size (folds) on the three evaluation metrics based on the JDT project for the HMOA algorithm.

lidity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity threats are related to the absence of similar work that uses bug localization technique to generate a dependency graph among several bug reports and therefore recommend those bugs in sequential order. For that reason, we compared our proposal with different mono-objective formulations that use one metric only like the score of bug priority. The developers were asked to evaluate different systems using different tools. We did not allow developers to evaluate different tools on the same system. The developers were distributed among the systems and tools based on their background/expertise to ensure almost the same level for all systems and tools. When each developer is asked to evaluate one different tool per system, we reduce the potential bias in the experiments since they are using the tools for the first time and they are exploring each time a new system. Our results show that the productivity has gotten better for the majority of our developers regardless of their experience and skills set.

External validity refers to the fact that our survey has been conducted by 29 developers with a variety of skills and number of experience. Thus, we can affirm that our results will hold its accuracy with a different set of developers with different level of expertise or knowledge. Also, time collection was left to each individual developer who manually noted the time they started and finished localizing a defect. This could have resulted in introducing error as every developer performed differently.

Finally, External validity could be related to the type of projects we used in the survey in which we used six different widely-used open-source systems belonging to the different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, other programming languages, and to other practitioners.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error, which is commonly used in the SBSE community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance.

3.6 Conclusion

We propose, in this thesis, an automated approach to localize and rank potential relevant methods for bug reports as an extension of our previous work limited to class level recommendations. Our approach finds a trade-off between minimizing the number of recommended methods and maximizing the correctness of the proposed solution using a hybrid multi-objective algorithm. The correctness of the recommended methods is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. Our approach uses the main steps, the first step finds the best set of classes satisfying the two conflicting criteria of relevance and number of classes to recommend using a global search based on NSGA-II. The second step is to locate the most appropriate methods to inspect, using a local multi-objective search based on Simulated Annealing (MOSA) from the list of classes identified in the first step.

This thesis presents the results of an empirical study with an implementation of our hybrid multi-objective approach based on 22,000 bug reports. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than state of the art techniques on 6 open source systems. As part of our future work, we plan to extend our work to consider the severity of the bugs when identifying relevant files. Furthermore, we are planning to address the problem of finding the qualified developers to fix the bugs based on the outputs of our bug localization approach. Finally, we will extend our work to handle multiple bugs reports at the same time and consider the dependency between them when recommending code fragments to the developers.

CHAPTER IV

Considering Dependencies Between Bug Reports to Improve Bugs Triage

4.1 Introduction

Software maintenance involves, typically, localizing and fixing a large number of defects that arise during development and evolution of systems *Zhang et al.* (2016). Localizing these software defects is expensive and time-consuming process which typically requires highly skilled and knowledgeable developers of the system. The localization process includes a manual search through the source code of the project in order to localize a single bug at a time *Jones* (2008). The number of these bug reports can be large. For example, MOZILLA had received more than 420,000 bug reports *Bettenburg et al.* (2008). These reports are important for managers and developers during their daily development and maintenance activities including bug localization *Fischer et al.* (2003). Due to the large number of reported bugs in successful projects, it is critical to efficiently manage them to improve developers productivity and quickly localize and fix these bugs *Zou et al.* (2018).

Each bug report has a set of attributes such as the bug summary/title, description, reported date, reporter or bug report creator and bug severity/priority as a scale that goes from P1 to P5. In general, developers and project managers have to analyze and assign a level of severity on each bug then assigning developers to localize and fix the bugs that are ranked based on their level of severity. This bug triage process plays an important role in software maintenance since the timely localization and correction of bugs are critical for the reputation of the organization and customers' satisfaction.

Once a bug report is assigned to a team, one of the developers uses it to reproduce the abnormal behavior to find the origin of the bug. However, the poor quality of bug reports can make this process tedious and time-consuming due to missing information. An efficient automated approach for locating and ranking important code fragments for a specific bug report may lead to improve the productivity of developers by reducing the time to find the cause of a bug *Fischer et al.* (2003).

Although several techniques have been proposed to localize bugs *Wong et al.* (2016); *Almhana et al.* (2016) and predict the severity of bugs *Uddin et al.* (2017); *Chaturvedi and Singh* (2012); *Zhang et al.* (2016), the existing studies related to the management of bugs report are mainly based on the priority scores to rank and assign bug reports without looking to the possible dependencies between them *Zheng et al.* (2006); *Canfora et al.* (2011); *Li et al.* (2006). Thus, developers may get assigned bug reports related to completely different files to be inspected which may increase the cognitive effort of the developers navigating between these independent bug reports. For instance, a developer may spend time understanding files A and B for Bug report B1 then he needs to check again these same files for bug report after working on three other independent bugs reports. We start, in this thesis, from the hypothesis that a better way to manage bugs reports is to group together those with a similar level of priorities and also sharing a common number of files to be inspected and fixed. In fact, several empirical studies show that the majority of bugs may not appear in isolation and they are related to each other *Zheng et al.* (2006); *Canfora et al.* (2011); *Li et al.* (2006). These dependent bug reports have several common files to inspect to localize the bugs.

To the best of our knowledge, we propose one of the first studies that consider the dependencies between bug reports in order to rank and group them while still considering their priorities. The proposed approach is mainly to validate the hypothesis that ranking and grouping bug reports based on the dependencies between them (classes to be inspected) besides the bugs priority can improve the productivity of developers and help them to localize bugs faster and more efficiently than considering them in isolation based only on the priority scores of the bug.

Our approach aims to find a trade-off between ranking the bug reports based on (1) their dependency and (2) their priority. The dependencies are extracted based on the list of files to be inspected from the bug report description using our previous bugs localization work *Almhana et al.* (2016) using a combination of lexical and history based measures. We selected that technique due to its high accuracy in localizing relevant files with over 80% in precision and recall. After extracting the list of files to inspect for each bug report, we adopted a multi-objective search, based on NSGA-II *Deb et al.* (2002), to find a trade-off between bugs priority and dependencies to rank the bug reports when assigned to developers. Thus, the manager or developer can select the best schedule of the bugs based on his/her preferences from the list of non-dominated ranking solutions generated by NSGA-II. For instance, a solution with high priority score and low dependency can be selected when the goal is to mainly focus on localizing the most severest bugs independently from the required effort. We selected NSGA-II algorithm since it is widely used in similar software engineering problems such as the next release problem *Geng et al.* (2018).

An experiment has been conducted to compare our approach with the only use of bugs priority to rank bug reports *Yu et al.* (2010); *Goyal et al.* (2015); *Xuan et al.* (2012); *Alenezi and Banitaan* (2013); *Lamkanfi et al.* (2011); *Kanwal and Maqbool* (2010). We conducted a pre-study and post-study survey to evaluate our the performance of our tool with participants based on 4 open source projects. The results show significant time reduction of over 30% in correctly localizing the bugs simultaneously comparing to the traditional bugs prioritization technique based on priority.

The remainder of this chapter is as follows: Section 2 is dedicated to describing the problem and our motivation to find a solution for it. Section 3 describes the proposed approach to localize bugs and then prioritize them. The evaluation of our approach and its results on several research questions with the answers and the discussions on those research questions are explained in Section 4. Finally, concluding remarks and future work is provided in Section 5.

4.2 Problem Statement

The bug triage process involves intensive time and resources in order to manage and analyze all reported bugs on a daily basis. Typically, project managers need to understand the reported bug, tweak the bug description and check for duplication, then assign priority or severity of a bug and finally assign it to a developer.

As of May 2019, the Mozilla bug database contains over 172,000 for only Firefox project only and the Eclipse bug database over 210,000 bug reports for Eclipse project only. On average, Mozilla received 212 and Eclipse 224 new bug reports on each week. Thus, clearly, the manual management defects for large software projects is not practical to prioritize and reank a large load of reported bugs. Furthermore, it is important to efficiently assign these bugs to reduce potential delays in localizing and fixing them.

Most of the existing work on the bugs prioritizing mainly focus on the assigned priority or severity to a bug either manually or automatically using static/dynamic analysis and the history of changes/bugs *Yu et al.* (2010); *Goyal et al.* (2015); *Xuan et al.* (2012); *Alenezi and Banitaan* (2013); *Lamkanfi et al.* (2011); *Kanwal and Maqbool* (2010). They treated bug reports in isolation despite that recent empirical studies show that a large number of simultaneous bugs were located on the same files *Zheng* *et al.* (2006); *Canfora et al.* (2011); *Li et al.* (2006). To the best of our knowledge, none of those techniques considered finding the dependencies among several bugs when ranking and grouping them to assign to developers. Recommending a list of bugs that share some common potential files to be inspected would be helpful to minimize the cognitive effort spent by a developer to jump from package to package or from file to file that are not related. Recent studies show that reducing such cognitive effort is a key to improve the productivity of developers working on multiple tasks *Zheng et al.* (2006); *Canfora et al.* (2011); *Li et al.* (2006).

Table 4.1 shows a list of 4 bug reports from the Eclipse Birt project that they were reported on Bugzilla within two days. By looking at the bugs description and their resolution on Github, we found that all of them are related to the core component/module of the software and require inspecting almost the same files and/or directory to localize and fix them. Typically, developers prefer to work on defects that are dependent on each other so that they can focus on one set of files rather getting disrupted with multiple not related bugs. Our hypothesis that the bug triage process will significantly save time and resources if we consider the dependencies between bugs as an additional criterion to the bugs severity.

4.3 Approach

4.3.1 Approach Overview

Our approach aims at exploring a large number of possible combination to find the best ranking of bug reports based on the dependency between them and their priority. The search space is determined not only by the number of possible dependencies between bug reports but also by the order in which they are proposed to the developer.

Our approach aims at exploring a large search space to find relevant classes to inspect by developers, given a description of a bug report. In fact, bug reports may

Bug ID	**Bug Summary**	**Bug Reported**	**Inspected Files**
Bug 456730	Missing default value in initializing scriptContext	2015-01-05	core/ org.eclipse.birt.core/ src/org/eclipse/birt/-core/.. /ScriptContext.java
Bug 456725	Optimize the performance of ULocale.forLocale	2015-01-05	core/ org.eclipse.birt.core/ src/org/eclipse/birt/-core/.. /LocaleUtil.java
Bug 456723	org.eclipse.birt. core.util.IOUtil doesn't check EOF	2015-01-05	core/ org.eclipse.birt.core/ src/org/eclipse/birt/-core/.. /IOUtil.java
Bug 456847	BirtDateTime function in chart's onRender function causes render failure	2015-01-06	core/ org.eclipse.birt.core/ src/org/eclipse/birt/-core/.. /CategoryWrap-per.java

Table 4.1: List of 4 bugs in Eclipse Birt project

require the inspection of more than one class to identify and fix bugs *Zheng et al.* (2006). Our previous work for bugs localization *Almhana et al.* (2016) is executed to identify relevant files to inspect for all the pending bug reports. The identified common files between the bug reports will represent the dependencies of all reported bugs we want to prioritize. Then, our bug prioritization component takes as input these dependencies along with the bug priority that has been assigned to each bug report. Our multi-objective search algorithm generates the best possible scheduling solutions to inspect the bugs to find a balance between priorities and dependencies of bugs. We represented the solution as a graph to guide developers to which bug needs to be resolved first, taking into consideration the two objectives of maximizing the number of files to inspect (maximize the intersection between consecutive bug reports in terms of files to inspect) and the bugs priority/severity that has been assigned manually by the users.

The general structure of our approach is sketched in Figure 4.1. It takes two inputs, the bug priority assigned by the user and recommended classes generated by the bugs localization tool (dependencies). The output is a set of non-dominated solutions of ranked bugs to inspect by the developer. Our heuristic-based optimization steps are formulated based on two main conflicting objectives. The first objective is to minimize the number of new classes to inspect between each pair of consecutively reported bugs. The second objective is to maximize the number of high priority bugs to be ranked first in the sequence of reported bugs. Thus, we consider, in this thesis, the task of prioritizing bugs as a multi-objective optimization problem using the non-dominated sorting genetic algorithm (NSGAII) *Deb et al.* (2002).

4.3.2 NSGA-II

In this thesis, we adapted one of the widely used multi-objective algorithms called NSGA-II *Deb et al.* (2002). NSGA-II is a powerful global search method stimu-

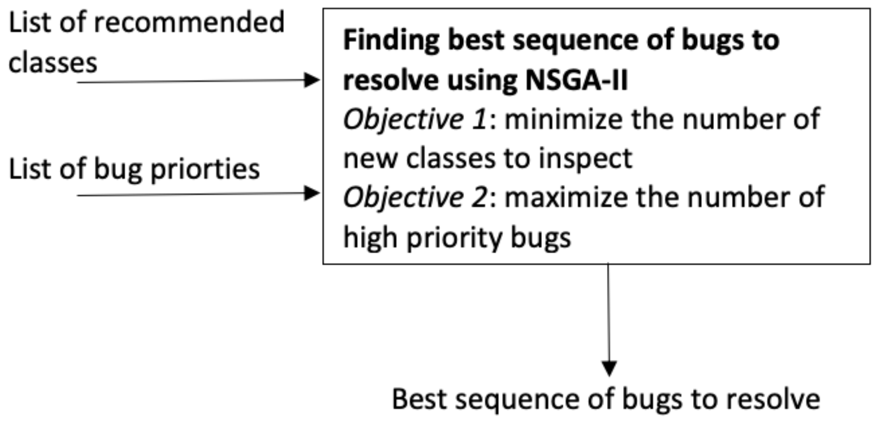

List of recommended classes
List of bug priorties
Finding best sequence of bugs to resolve using NSGA-II
Objective 1: minimize the number of new classes to inspect
Objective 2: maximize the number of high priority bugs
Best sequence of bugs to resolve

Figure 4.1: Approach Overview

lated by natural selection that is inspired by the theory of Darwin. We selected this multi-objective search algorithm since it was used for similar problems in software engineering *Almhana et al.* (2016); *Geng et al.* (2018); *Ramirez et al.* (2019).

The basic idea of NSGA-II is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. As described in Algorithm 1, the first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P0 using genetic operators such as crossover and mutation (line 2). Both populations are merged into an initial population R_0 of size N (line 5). As a consequence, NSGA-II starts by generating an initial population based on a specific representation that will be discussed later, using the exhaustive list of bugs from the bug reports to resolve given as input. Thus, this population stands of a set of solutions represented as sequences of defects to resolve, which are randomly selected and ordered *Almhana et al.* (2016).

The whole population that contains N individuals (solutions) is sorted using the

Algorithm 1 High level pseudo code for NSGA-II

```
1: Create an initial population $P_0$
2: Create an offspring population $Q_0$
3: $t = 0$
4: while stopping criteria not reached do
5:     $R_t = P_t \cup Q_t$
6:     F = fast-non-dominated-sort($R_t$)
7:     $P_{t+1} = \emptyset \ and \ i = 1$
8:     while $\mid P_{t+1} \mid + \mid F_i \mid \leqslant N$ do
9:         Apply crowding-distance-assignment($F_i$)
10:        $P_{t+1} = P_{t+1} \cup F_i$
11:        $i = i + 1$
12:    end while
13:    $Sort(F_i, \prec n)$
14:    $P_{t+1} = P_{t+1} \cup F_i[N - \mid P_{t+1} \mid]$
15:    $Q_{t+1}$ = create-new-pop($P_{t+1}$)
16:    t = t+1
17: end while
```

dominance principle into several fronts (line 6). The dominance level becomes the basis of a selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance to make the selection (line 9). This front F_i to be split, is sorted in descending order (line 13), and the first $(N - |P_{t+1}|)$ elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover, and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4) *Almhana et al.* (2016). The following subsections describe more precisely our adaption of NSGA-II to the bugs triage problem.

4.3.3 Solution Representation

Figure 4.2 shows a simplified representation of a solution (recommended schedule of bugs to resolve) generated by our web-based tool for bugs selected randomly from the bug repository (Bugzilla website) of Eclipse Birt project. This solution represents

Bug 456730	Bug 456725	Bug 456723	Bug 456847

Figure 4.2: A simplified example of solution representation

a possible sequence to resolve the reported bugs in Table 4.1 for Eclipse Birt project. The recommended classes of those defects share the same package or directory (core/ org.eclipse.birt.core/ src/ org/ eclipse/ birt/ core) that needs to be inspected by programmer. Thus, we group those defects together and recommend this cluster of bugs to one developer to resolve as a sequence. This simplified representation may not be sufficient to show the dependencies between the bug reports thus we adopted a graph-based representation that can be visualized to the users.

Figure 4.3 shows the sequence of bugs suggested by our web-based tool for 10 pending bugs selected randomly from the bug repository (Bugzilla website) of Eclipse Birt project. The user can interact with the graph to interactively change the proposed schedule and choose one path among several proposed paths (solutions). The different bugs scheduling solutions that can be explored by the developers or managers are represented in Figure 4.4 balancing the two objectives of severity and dependencies.

4.3.4 Fitness Functions

There are two fitness functions used in our multi-objective search based algorithm. The first fitness function measure encourages keeping high priority bugs first in a sequence and low priority bugs last in a sequence. The first fitness function is to maintain low cognitive effort between each pair of consecutively reported bugs. Our goal is to minimize as much as possible the number of new classes to inspect when the developer moves from one bug to the next consecutive bug in the sequence. Equation 4.1 preserves the level of dependencies between each pair of consecutive bugs, a higher value represents high similarity in dependencies (recommended classes) among bug reports. The objective of the formula is to maximize the intersection (in number of

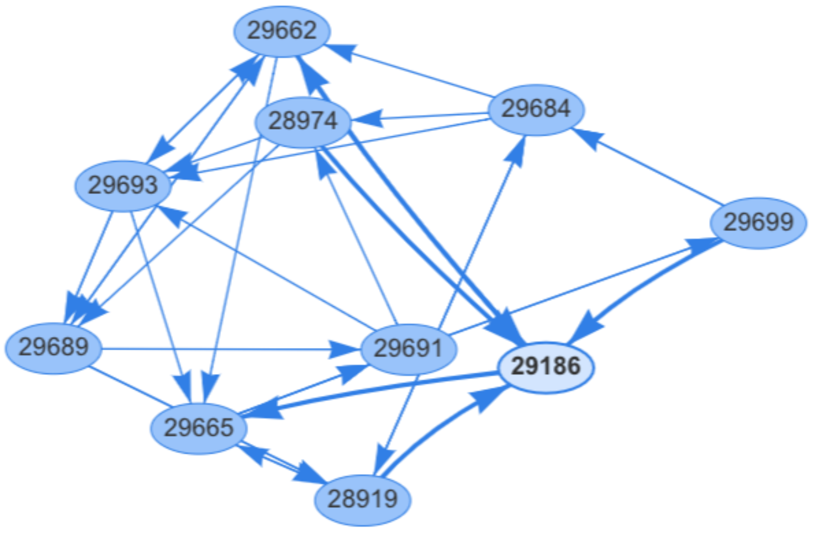


Figure 4.3: A graph that shows the order of each recommended solutions generated by our web-based software for particular set of pending bugs in Eclipse Birt Project

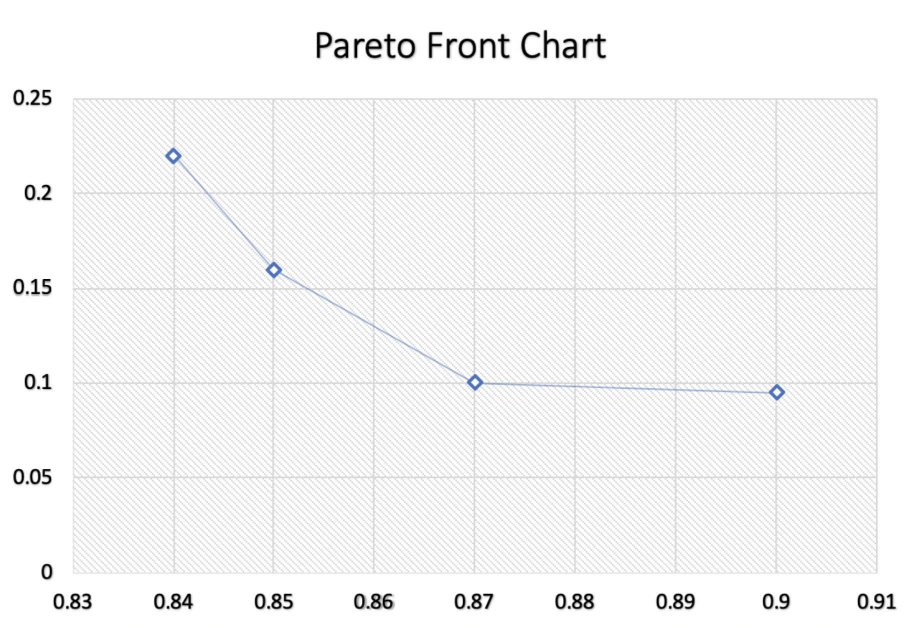


Figure 4.4: The Pareto Front of recommended solutions generated by our web-based software for pending bugs in Eclipse Birt Project to balance both severity (X-Axis) and dependencies (Y-Axis).

inspected files) between two consecutive bug reports. $NumFiles_{i,i+1}$ represents the total number of distinct files to inspect for bug(i) and bug(i+1). Bug(i) represents the set of classes that are related to bug (i) and similarly Bug(i+1) represents the set of classes that are related to bug (i+1), where (n) represents the number of bugs.

$$f1 = \sum_{i=1}^{i=n} \frac{Bug_i \bigcap Bug_{i+1}}{NumFiles_{i,i+1}} \tag{4.1}$$

The objective of the second fitness function is to minimize the differences between the priority of bug reports and the order of recommendations to solve the reported bug reports. Equation 4.2 calculates the difference in priority for a bug between the bug report and the recommended solution. We build a vector of reported bugs and sort them based on the priority value reported in the bug report. Then, we compare the position of a reported bug B_i in recommended solution with the position of the same bug B_i in the original order of reported bugs which is based on priority value reported on bug reports. Equation 4.2 calculates the difference between the priority value in bug report and the priority value in the recommended solution which is the position of the bug in the solution vector. Equation 4.2 calculates the sum of differences in priority between bug report and the recommended solution for each of the bugs where (n) represents the number of bugs.

$$f2 = \sum_{i=1}^{i=n} |IndexOfBug_{i,solution} - IndexOfBug_{i,report}| \tag{4.2}$$

The above two objectives are conflicting since minimizing the number of new classes to inspect between each pair of consecutively reported bugs may lead to resolving some low priority bugs however the scheduling solution may improve the overall productivity.

Table 4.2 shows the Pareto Front sketched by our web-based tool for 10 bugs

No.	Solution	Objective 1	Objective 2
1	28974, 29186, 29665, 28919, 29684, 29662, 29693, 29689, 29691, 29699	0.84	0.22
2	28974, 29693, 29689, 29662, 29665, 29691, 29699, 29684, 28919, 29186	0.85	0.16
3	29699, 29684, 28974, 29689, 28919, 29665, 29691, 29693, 29662, 29186	0.87	0.10
4	29699, 29186, 29662, 29689, 28919, 29684, 29693, 29665, 29691, 28974	0.90	0.095

Table 4.2: Pareto Front Results

selected randomly from bug repository (Bugzilla website) of Eclipse Birt project. This is an example of Pareto Front results (the recommended solutions) generated by our web-based software for particular set of bugs in Eclipse Birt Project.

4.3.5 Change Operators

In a search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards better solutions. We randomly select individuals for mutation and crossover. The probability to select an individual for crossover and mutation is directly proportional to its relative fitness in the population. In each iteration, we select half of the population in iteration i. These selected individuals will give birth to another half of the population of new individuals in iteration $i + 1$ using a crossover operator. Therefore, new two-parent individuals are selected for next iteration/generation.

The one point crossover operator allows creating two offspring P_1 and P_2 from the two selected parents P_1 and P_2. It is defined as follows: a random position, k, is selected. The first k bugs of P_1 become the first k elements of P_1. Similarly, the first k bugs of P_2 become the first k elements of P_2. Our crossover operator could create a child that contains redundant recommended bugs. In order to resolve this problem, for each obtained child, we verify whether there are redundant bugs or not. In the case of redundancy, we do not apply crossover operation on this particular bug.

An example of crossover operation, consider there are (2) vectors of recommended solutions as follows:

Solution 1 $\rightarrow$ (bug A, bug B, bug C, bug D, bug E)

Solution 2 $\rightarrow$ (bug F, bug G, bug H, bug I, bug J)

After applying crossover operator on both solutions, the outcome will be as follows:

Solution 1 $\rightarrow$ (bug A, bug B, bug H, bug I, bug J)

Solution 2 $\rightarrow$ (bug F, bug G, bug C, bug D, bug E)

4.4 Evaluation

In order to evaluate our approach for prioritizing multiple defects for developers, we conducted a human validation to evaluate the benefits of our work. The experiments included a pre-study survey to gather some personal information and technical background of the participants then a post-study survey to gather developer's feedback about our tool with some insights about future improvements to the tool. The obtained results are subsequently statistically analyzed with the aim to compare our multi-objective approach with three other approaches. The first approach is a traditional bug priority based approach and the second one is based on the dependencies between bug reports without considering the score of the priority reported in the bug report. The third approach is based on a first come first served resolution based approach. In this section, we present our research questions followed by experimental settings and parameters. Then, we discuss our results for each of the research questions. The data related to our experiments can be found in the following link *Almhana and Kessentini* (2020a)

4.4.1 Research Questions

In our study, we wanted to assess the performance of our approach by finding out whether it could identify the most appropriate sequence of bugs to resolve by developers. In order to examine our web-based software prioritization tool, we explored two primary research questions outlined below. The goal of this experiment is to check whether our proposed approach can propose a meaningful sequence of defects in which developers can localize and fix related bugs quickly and therefore companies can save some efforts in terms of time, resource and cost to make their systems more responsive to most recent bug reports. To this end, we defined the following research questions:

- RQ1: (Effectiveness) To what extend can the proposed approach recommend an appropriate sequence of bugs to resolve by developer?
- RQ2: (Comparison to other techniques) How does our approach perform compared to typical bugs management techniques?

The goal of RQ1 is to measure the effectiveness of our approach by calculating three different metrics mentioned in this paper whereas the RQ2 aims to compare our approach with other approaches to measure the effectiveness compared to three other approaches (FCFS, 2 mono-objectives approaches).

To answer RQ1, we evaluate the effectiveness of the recommended order of bugs to resolve by programmers. The effectiveness is evaluated by measuring the following metrics:

- **Number of Bugs** denotes the number of bugs that one individual developer can resolve within a time frame. The goal for this measure is to maximize the number of bugs that developer can finalize in order to have better productivity.

- **Resolution Time** denotes the time spent by developer to understand, identify, and resolve a single particular bug. Our goal is to minimize this measure in order to save resource cost.

- **Disruption Cost** measures the cost of transition time that developer may spend between each pair of bugs. Our approach aims to minimize this cost by recommending most related sequence of bugs.

To answer RQ2, we compared, using the above metrics, the performance of our multi-objective approach with first come first serve approach. Furthermore, we implemented two mono-objective formulations. The first one is a mono-objective algorithm with the only objective of bug priority score and a second one is a mono-objective algorithm with the only objective of bug dependency. Disruption cost means the time in which a developer spends to make the transition between one bug to another unrelated bug. This transition involves the time to change the developer's focus to understand the information given to the developer in the new bug and the time to examine the files related to the new bugs. This disruption cost is important because it can show the cognitive effort required by developers to move from one bug to the other when they are not related. Equation 4.3 formulates the distribution cost where n is the number of bugs to resolve. To best of our knowledge, there is no similar prior work to compare with that uses currently similar objectives of our approach.

$$DisruptionCost = \sum_{i=1}^{i=n}(|EndTimeBug_i - StartTimeBug_{i+1}|) \tag{4.3}$$

One way to show if the two objectives are conflicting is to compare the performance of the multi-objective search with a mono-objective formulation (aggregation of all the objectives). The comparison between a multi-objective technique with a mono-objective one is not straightforward. The multi-objective technique returns a set of non-dominated solutions while the mono-objective technique one returns a single optimal solution. To this end, we choose the nearest solution to the Knee point *Deb et al.* (2002) (i.e., the vector composed of the best objective values among the population members) as a candidate solution to be compared with the single solution returned by the mono-objective algorithm.

The knee point represents the maximum trade-off between the objectives thus it is reasonable to compare it with a mono-objective solution with equal weights of the different objectives aggregated in one fitness function. The fact that we are comparing a mono-objective formulation with equal weights to a knee point (representing the maximum possible trade-off) ensures a fair comparison. We used the knee point method as recommended by the current literature *Keller* (2019); *Emmerich and Deutz* (2018); *Deb and Gupta* (2011)

Both surveys (pre-study and post-study questionnaire) were conducted on twenty-nine developers who have a variety of skills and expertise. Table 3.2 shows a list of six open source systems that developers use in the experiment. The survey tells us whether our approach was successful to save cost and time in resolving bugs.

4.4.2 Software Projects and Experimental Setting

As described in Table 3.2, we used six open-source systems:

- **Eclipse UI** is the user interface of the Eclipse development framework.
- **Eclipse Jetty** is a Java HTTP server and Java Servlet container.
- **Eclipse AspectJ** is an aspect-oriented programming (AOP) extension created for the Java programming language.
- **Eclipse Birt** provides reporting and business intelligence capabilities.
- **Eclipse SWT** is a graphical widget toolkit.
- **Eclipse JDT** provides a set of tool plug-ins for Eclipse.

Project	# Bugs	# Resolved Bugs	# Developers	Average Resolution Time	Time Frame
Eclipse UI	84,136	57,251	778	89 days	Oct-2001 to May-2019
Eclipse Birt	23,218	19,452	154	40 days	Jan-2005 to May-2019
Eclipse JDT	58,822	34,050	272	52 days	Oct-2001 to May-2019
Eclipse AspectJ	3021	2270	22	38 days	Sep-2002 to May-2019
Eclipse Jetty	3813	1184	14	43 days	Mar-2009 to May-2019
Eclipse SWT	24,049	19,559	184	61 days	Oct-2001 to May-2019

Table 4.3: Studied Projects

Project	# of Developers	Avg. # of Experience
AspectJ	15	9.5 years
Birt	18	7 years
SWT	15	10 years
Jetty	14	10.5 years
Eclipse UI	24	6 years
JDT	21	6.5 years

Table 4.4: List of developers participated in the experiment and their disruption among several projects along with the number of years of experience

Table 3.2 shows the different statistics of the analyzed systems including the time range of the bug reports, the number of bug reports, the number of closed and resolved bugs in a project, the number of developers involved with project and the average of time spent to resolve a bug and close its corresponding bug report. The total number of collected unresolved bug reports is about 63,000 bug reports for the six open source systems. All these projects are using BugZilla tracking system and GIT as a version control system.

4.4.3 Meta-heuristic Parameters Tuning

An often-omitted aspect in meta-heuristic search is the tuning of algorithm parameters. In fact, parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and each system, we performed a set of experiments using several population sizes: 10, 20, 30, 40 and 50. The stopping criterion was set to 100,000 fitness evaluations for all search algorithms in order to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion since our approach requires multiple objectives. Each algorithm was executed 30 times with each configuration and then the comparison between the configurations was performed based on different metrics described previously using the Friedman test. The other parameters values were fixed by trial and error and are as follows: (1) crossover probability = 0.4; mutation probability = 0.3 where the probability of gene modification is 0.1. *Almhana et al.* (2016)

4.4.4 Results

4.4.4.1 Results for RQ1

For this research question, we examined the number of bugs that the developers were able to resolve within the 2-hour window. Figure 4.5 shows the difference in performance between our multi-objective approach and the first come first serve approach. Furthermore, we measured the performance of the mono-objective approaches by considering sperately the score of bug priority or bug dependency. The results show that the multi-objective combining the benefits of both mono-objective approaches are presenting much better results in terms of fixing bugs.

Figure 4.6 describes the average time spent by the developer to resolve one single defect in a certain project. This figure shows the difference in the number of minutes between our multi-objective approach and other three different approaches such as first come first serve, bug priority, and bug dependency approach. We found that the familiarity with the associated files to a bug play an important factor in the time that the developer may spend on one individual bug which explains the signficant outperformance of our approach.

Figure 4.7 presents the disruption cost or cognitive efforts needed to completely shift from one bug to another. We found that this cost is too high in First Come First Serve (FCFS) and medium in Bug Priority but it drops significantly in Bug Dependency or multi-objective approach which shows the benefit of considering bugs dependency to improve the productivity of the developers.

To conclude, it is clear that the multi-objective approach significantly reduce the efforts spent by the developers to fix bugs when they are ranked based on a combination of their dependency and priority.

4.4.4.2 Results for RQ2

Figure 4.5 and Figure 4.6 confirm the efficiency of our multi-objective approach over other techniques used to prioritize bug reports based on severity or first come first served. In Figure 4.5, our approach shows an average of 3 defects in 2-hour window for all evaluated projects whereas first come first serve (FCFS) and Bug Priority approach shows an average of 1 defect in a given time window. Bug Dependency technique produces a promising result with an average of 2.5 which is very close to multi-objective approach's outcome and that is due to the importance of recommending the bugs that share the same set of files/classes to inspect. The complexity of the project plays an important role in localizing and fixing bugs, developers localized and fixed 2 to 3 bugs in Eclipse UI or JDT projects as opposed to 5 bugs in Jetty.

In Figure 4.6, the multi-objective approach has as low as 21 minutes and as high as 78 minutes on average to resolve a single defect. Bug dependency comes next in efficiency after the multi-objective approach with a low of 28 and high of 67 minutes. The third approach is Bug Priority with unremarkable results of 78 minutes on average. FCFS result is considered the worst with an average of 123 minutes since it does not follow any dynamic strategy in choosing the next bug in line to resolve. We noticed a big gap between FCFS and others as FCFS does not consider the complexity, size, severity, and urgency of the bug but rather goes from one bug to another. Our approach helps to reduce the resolution time even in the large and complicated systems, 187, 176, and 154 minutes were recorded for FCFS in Birt, Eclipse UI, and JDT respectively and 66, 44, and 78 minutes were recorded in multi-objective approach for those same projects.

Figure 4.7 shows an average of 6 minutes in multi-objective and 8 minutes in Bug Dependency approach. One of the reasons that make the localization and fixing time too high in FCFS is the high disruption time of 39 minutes on average. Bug Priority does slightly better than FCFS with 22 minutes but Bug Priority is still far away from Bug Dependency or multi-objective approach. Furthermore, we noticed that the disruption cost increases when the size of the project becomes larger. Birt is an example of a large project which required 10 minutes of disruption cost whereas it is around 5 minutes for other smaller projects like Jetty.

To conclude, the proposed multi-objective approach outperforms mono-objective ones which confirm the need to consider bugs dependencies when scheduling them to be repaired by developers.

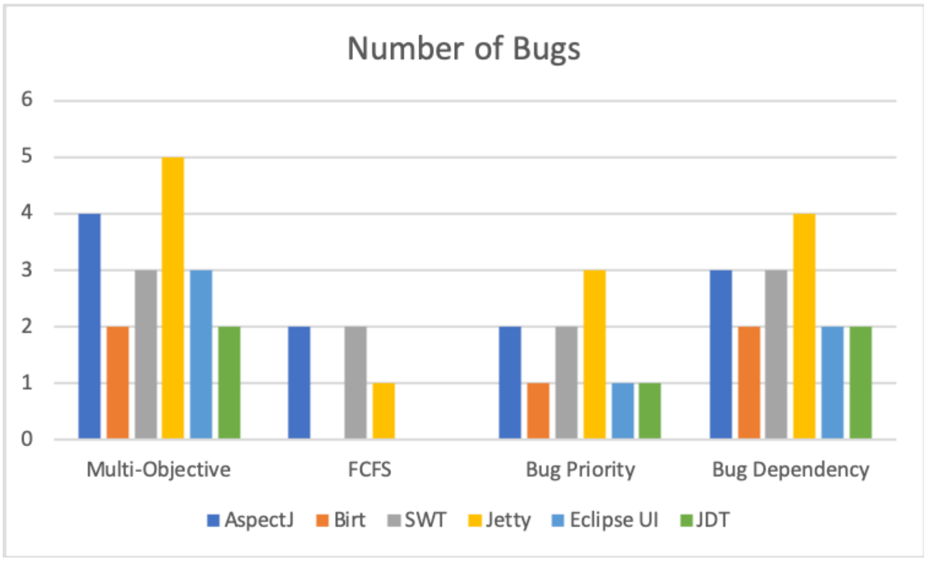


Figure 4.5: Comparison of number of bug to resolve a particular bug using our prioritization tool versus FCFS tool along with two of mono-objective approaches for each of the six tools

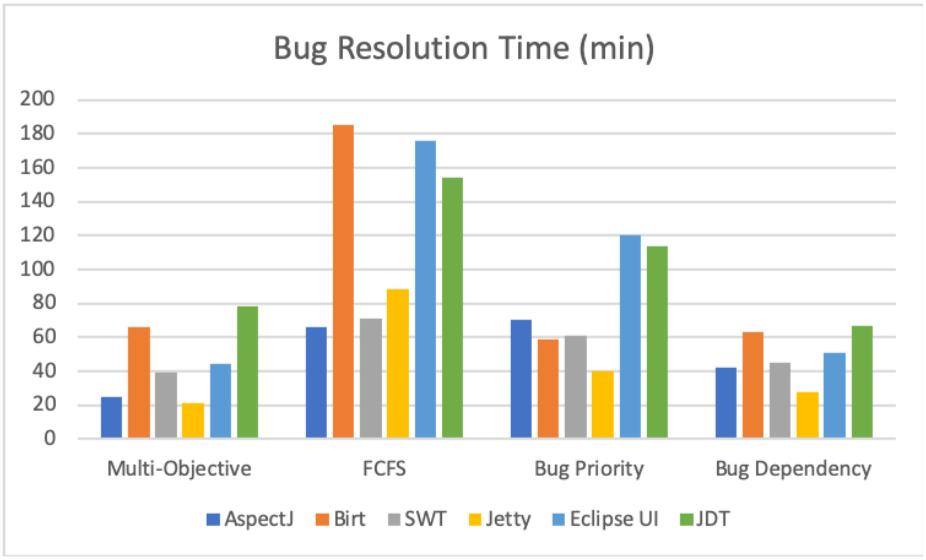


Figure 4.6: Comparison of average time spent to resolve a particular bug using our prioritization tool versus FCFS tool along with two of mono-objective approaches for each of the six tools

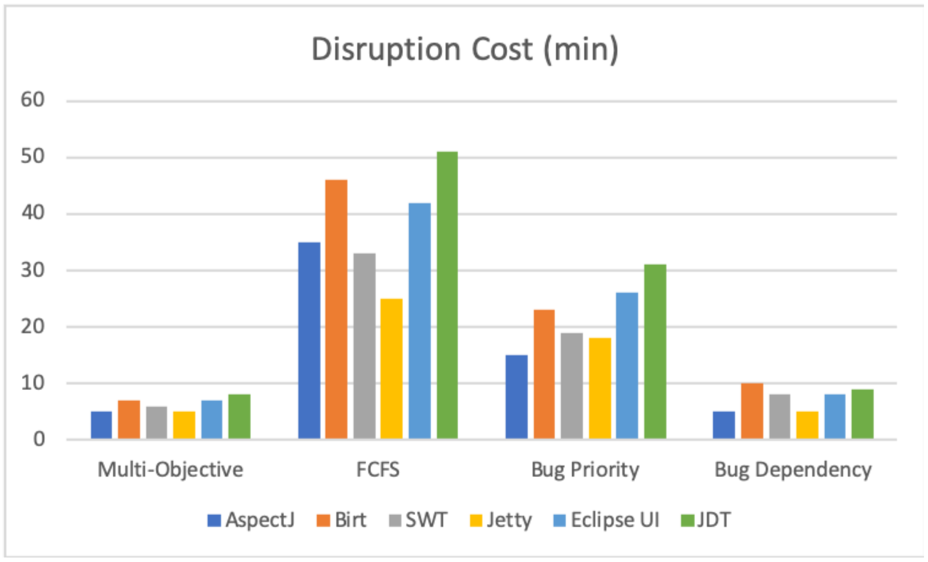


Figure 4.7: Comparison of disruption cost to transit from one bug to another using our prioritization tool versus FCFS tool along with two of mono-objective approaches for each of the six tools

4.4.4.3 Pre-study Survey Results

All the participants have a job in industry as software engineer or technical lead. 87% of our participants hold a bachelor degree in computer science, Table 4.4 shows the list of six (6) open source software used in the study along with the number of developers who participate in each of those projects with average years of experience of those participants. Figure 4.8 shows the distribution of expertise for our participants regarding the 5 different categories listed in the questionnaire. 16 participants were working on software testing and bug repair tasks as part of their regular duties, which was one of the main criteria used to solicit their participation, based on our previous collaborations and contacts.

4.4.4.4 Post-study Survey Results

Chart 4.9 shows the results we gathered from our participants about the three post-study survey's questions. For Q1, we found that 72% thought that the recommended solution (the order of resolving the bugs) made the whole task easier than

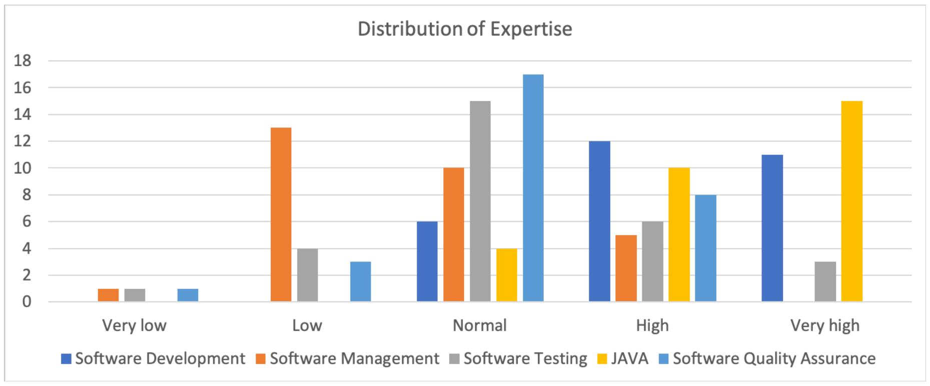


Figure 4.8: Distribution of Expertise for the participants in the pre-study survey

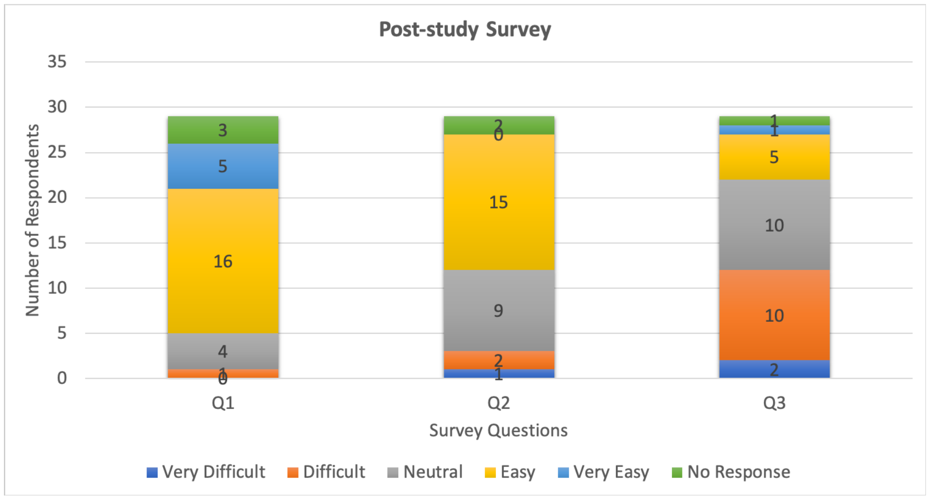


Figure 4.9: Post-study survey results

normal. For Q2, the majority, over 50% found that the new approach tends to save developers' time to localize bugs and resolve. For Q3, we found that our participants have noticed the difference between First Come First Serve (FCFS) and our approach in which 12 developers reported that task was difficult, and 10 developers found it neutral where they did not notice any improvements.

4.5 Threats to Validity

We want to acknowledge several threats to the validity of the paper such as the factors that can bias our empirical study. These factors can be classified into three categories: internal validity, construct internal, and external validity. Construct validity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity threats are related to the absence of similar work that uses bug localization technique to generate a dependency graph among several bug reports and therefore recommend those bugs in sequential order. For that reason, we compared our proposal with different mono-objective formulations that use one metric only like the score of bug priority. The developers were asked to evaluate different systems using different tools. We did not allow developers to evaluate different tools on the same system. The developers were distributed among the systems and tools based on their background/expertise to ensure almost the same level for all systems and tools. When each developer is asked to evaluate one different tool per system, we reduce the potential bias in the experiments since they are using the tools for the first time and they are exploring each time a new system. Our results show that the productivity has gotten better for the majority of our developers regardless of their experience and skills set.

External validity refers to the fact that our survey has been conducted by 29 developers with a variety of skills and number of experience. Thus, we can affirm that our results will hold its accuracy with a different set of developers with different level of expertise or knowledge. Also, time collection was left to each individual developer who manually noted the time they started and finished localizing a defect. This could have resulted in introducing error as every developer performed differently.

Finally, External validity could be related to the type of projects we used in the survey in which we used six different widely-used open-source systems belonging to the different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, other programming languages, and to other practitioners.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error, which is commonly used in the SBSE community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance.

4.6 Conclusion

We proposed an approach for bugs management by taking into consideration both the severity and dependencies between reports. Our solution is based on the use of multi-objective search to find a trade-off between these two conflicting objectives. The validation of our work shows that there were significant time savings when developers inspected bugs comparing to existing methods treating each bug individually as first come first serve or relaying on priority scores only.

As part of our future work, we envision the extension of this approach to improve the bugs management process by recommending developers to be assigned for bugs based on their background and prior expertise. The users can interact more with the suggested recommendations in order to update the assignments. In addition, we are planning to extend our current work with multiple other bug repository systems beyond Bugzilla. We also would like to validate the proposed tool on proprietary software systems to generalize the obtained results.

CHAPTER V

Understanding and Characterizing Changes in Bugs Priority: The Practitioners' Perceptive

5.1 Introduction

A bug is a software defect that causes abnormal or erroneous behavior according to functional or non-functional requirements (such as security and performance) *Chung et al.* (2000); *Zaman et al.* (2011); *Guo et al.* (2010); *Zimmermann et al.* (2012); *Shihab et al.* (2013). Different bugs impact the software system differently based on the degree of severity associated with each bug *Chaturvedi and Singh* (2012); *Lamkanfi et al.* (2011); *Xia et al.* (2014). Therefore, it is critical to efficiently manage bugs priority *Tian et al.* (2013); *Kumari and Singh* (2018); *Kanwal and Maqbool* (2010); *Yu et al.* (2010); *Tian et al.* (2015). In this context, the effectiveness of assigning priorities becomes important—if priorities of a fair number of bugs are changed, it indicates delays in fixing critical bugs *Tian et al.* (2013); *Kumari and Singh* (2018); *Kanwal and Maqbool* (2010); *Yu et al.* (2010); *Tian et al.* (2015).

Several studies explored methods to predict bugs priority in software systems *Tian et al.* (2013); *Yang et al.* (2014); *Sharma et al.* (2012); *Kumari and Singh* (2018); *Tian et al.* (2015); *Zaman et al.* (2012); *Uddin et al.* (2017). To the best of our knowledge, there has been little prior work on understanding the dynamics of changing bug priorities. Understanding changes in bugs priority can help us to quickly fix severe bugs and avoid delays, identify areas that need tool support for automated validation of bug priority change requests and better documentation of these changes. The goal of this thesis is to characterize the overall change process of bugs priority.

We advocate that a critical and fundamental step in providing an efficient support for manager and developers to enable them validating bugs priority change is to understand the bugs priority dynamics; it involves discover and characterize *Why* and *When* bug priorities change, and *Who* performs the change. Thus, the primary goal of this thesis is to observe and understand the changes in bugs priority in order to build a 3-W (Why, When, and Who) model. In this pursuit, we used two complementary methods in our study. As a first step, we discovered insights about the rationale of bug priority changes, their frequency, and when/why these changes were observed by interviewing 11 software developers, managers, and executives from eBay as part of a funded project. We established an initial model for characterising changes in bug priority as an outcome of this first step. In a second step, we performed a survey with an additional 38 developers to enquire about their experiences with finding, validating, and documenting the changes in bugs priority. During these two steps, we answered the following three research questions (RQ).

RQ1: *Why does priority of a bug change?*

RQ2: *Who changes the priority of a bug?*

RQ3: *When does the priority of a bug change?*

We propose a 3-W bugs priority change model obtained from the interviews and the survey. We have also conducted a manual inspection of X bugs reports, developers' comments, and source code changes from Y open source systems to compare the final 3-W bugs priority changes model with actual bugs priority changes extracted from open-source projects to answer the following research question.

We have also compared in this thesis the experience of developers in finding, validating, and documenting bugs priority changes with samples of actual ones reveal areas for improvement. We found that developers indeed change the priority of bugs multiple times. Table 5.1 shows an example of a bug report with its log of activities that show the number of times developers change the priority along with the bug priority both before and after the change.

Our 3-W model suggests the following rationale of changes in bugs priority: 1) lack of time to complete the task, 2) the category of the bug such as security related bugs or functionality related bugs or user interface related bugs, 3) the type or the domain of project such as security related projects, desktop application or web-based application, a plugin tool or standalone program, 3) dependencies to other bugs, 4) lack of understanding or misunderstanding the bug report, and 5) accidental changes by mistake. In addition, we also found that some developers do not follow "ethical" practices while changing bugs priority. For instance, they may reduce bugs priority just to bypass quality gates and to release code quickly.

Our findings can enable (1) researchers to automatically validate bugs priority changes and understanding their rationale, (2) educators to teach and emphasize the management of bugs and prioritize software maintenance activities, and (3) practitioners to use a standard format for documenting and discussing changes in bugs priority. Though we identified a set of essential components of changes in bugs priority, adoption of the components remains context-dependent in practice. Using our model, software development teams can design their organization-specific guidelines to include or exclude the proposed components for validating changes in bugs priority.

The primary contributions of this thesis are as follows:

1. A detailed model to understand changes in bugs priority based on the perspective of practitioners.

2. A study of the experiences of software developers requiring, finding, and documenting changes in bug reports.

Who	**When**	**Before**	**After**	**Attributes Changed**
User1	2008-05-11	P3	P5	-
User2	2008-05-20	P5	P3	Target Milestone
User3	2008-08-03	P3	P5	-
User2	2008-08-21	P5	P3	Status & Resolution
User1	2009-05-18	P3	P5	Target Milestone
User2	2009-05-26	P5	P3	Target Milestone

Table 5.1: Bug Report# 221310 from BIRT project to show the activities that happened on the priority of this bug report.

3. Investigation of areas for improvement in current practices of changing bugs report. Identifying gaps will provide valuable recommendations for researchers to develop tools for validating changes in bug reports and developers and managers wanting to improve their current pipeline to manage bug reports.

Replication Package. All material and data of the bugs report used in our study as well as the developers' anonymized answers are available in our replication package *Almhana and Kessentini* (2020c).

The remainder of this chapter is as follows: Section 2 describes the design of our empirical study including the research questions. The results are described in Section 3. Finally, concluding remarks and future work are provided in Section 4.

5.2 Approach: Study Design

Our study aims to understand the rationale of changes in bugs priority and therefore will guide the automation of validating change requests by developers. As described in Figure 5.1, we first used unstructured interviews with 11 developers, managers, and executives from eBay, as part of a funded project, to discover the rationale of changes in bugs priority and thereby, design an initial model. These interviews allowed rich conversations and insights. These brainstorming sessions helped us to establish the initial thoughts in characterizing the changes in bugs priority to discover the reasons behind changes the priority, the time of changing the priority, and the individuals who perform the changes. Then, we extended the obtained initial documentation model with a larger number of 38 practitioners using a survey. The practitioners answered our questions about their experiences in performing, finding, and documenting these bugs priority changes. Finally, we conducted a quantitative validation to compare the outcomes of the interviews and survey with actual changes in bugs priority extracted from X open-source systems. The use of these mixed methods has been widely employed by several other studies of software developers *Codoban et al.* (2015); *Hilton et al.* (2017); *Easterbrook et al.* (2008); *Tao et al.* (2012).

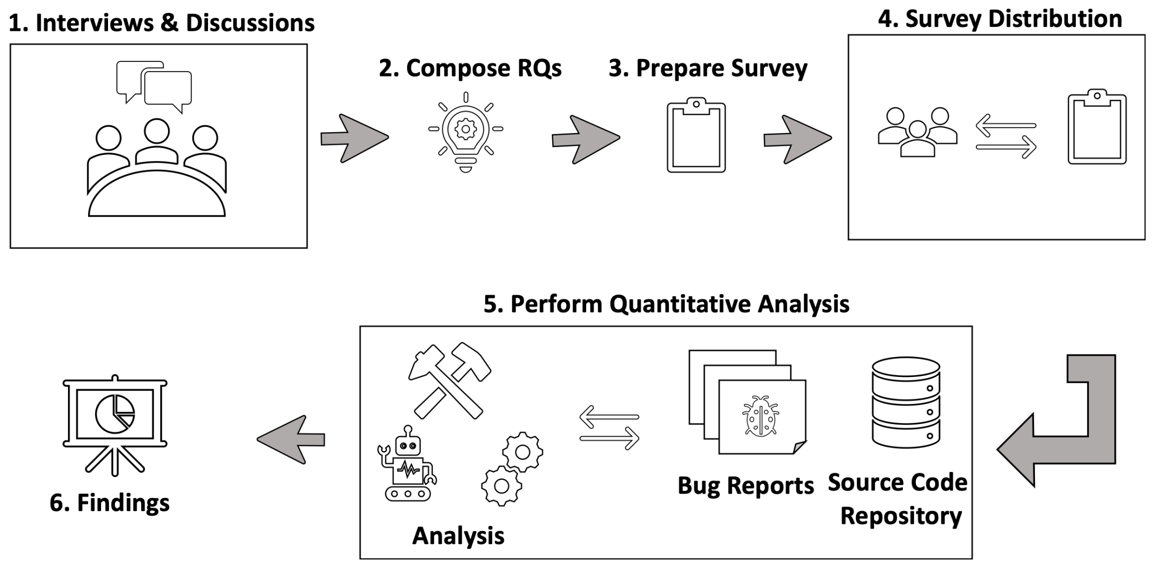


Figure 5.1: Study design

5.2.1 Research Questions

We defined the following main three research questions.

● **RQ1**: *Why does the priority of a bug change?* The aim of this first research question is to gather exhaustive possible reasons behind changing the priority of bugs. An understanding of the rationale will help (a) practitioners to better document priority changes of bugs; and (b) researchers to build tools for automatically checking the requests of changing bug's priority.

● **RQ2**: *Who does change priority of a bug?* The aim of this research question is to identity the main stockholders who change the priority of bugs and their role in the team or in the project (e.g. tester, manager, and owner of the bug). The outcome of this research question can help us to understand the needs for changing the priority based on the role of the people who made the change.

● **RQ3**: *When does priority of a bug change?* Since there are no restrictions, in general, on when bugs priority can change, the aim of this research question is to study the possible correlations between the dates of priority changes and bug's creation, release date, or the date of assigning a developer on the bug. The outcomes of this research question may inform us about temporal patterns to change bugs priority in suspicious time such as new release deadline.

To summarize the outcome of this contribution, our aim is to compare the outcomes of both the interviews and the survey with actual changes of bugs priority extracted from open source systems. Observations from the comparison between the practitioners' need and actual priority changes found through a quantitative analysis could lead us to the areas of improvement for researchers and practitioners to address them.

5.2.2 Phase 1: Interviews with Developers to Design a Model for Bugs Priority Changes

5.2.2.1 Interview setup

The goal of this first phase of our research was to build a model to characterize the dynamics related to bug's priority, to understand why and when they happen, and to identify the individuals' role who make the change or get impacted by the change.

Prior to starting the interview sessions, we performed an in-depth analysis of previous studies that are related to bug's priority changes or bug's priority in general. We list them in Table 5.2 and discuss them in detail in the related-work section. The aim of this in-depth analysis is to understand the current state-of-the-art and to gather insights on *WHY* priorities change, *WHEN* do they change, and *WHO* makes the change. The majority of existing literature focuses on bug's priority prediction and, in some cases, recommending developers to fix bug reports. To the best of our knowledge, none of them analyzed the historical changes in priority levels to build a model that can validate the accuracy of changes in priority made by developers. Typically, textual and temporal components of a bug report, author's information with historical bug reports are the four most used components or metrics for predicting bug's priority. We considered these observations from the current literature about bugs priority in our unstructured interviews with eBay developers, managers, and executives.

5.2.2.2 Participants Selection

We advertised our study in mailing lists that covered developers from many industrial partners, including those who collaborated with us in the past, to validate our approach in characterizing the overall change process of bugs priority. We interviewed 11 participants, after eliminating three other practitioners because they were not able to attend the whole interview session and provided very limited and quick feedback in the discussion.

5.2.2.3 Interview with selected volunteers

We started the face-to-face interviews by providing examples of several bug-tracking systems such as Jira [1] and BugZilla [2]. We presented the process of logging a bug report and setting the bug's priority. Then, we showed the mechanism by using several examples of changing the priority in those bug-tracking systems. We also demonstrated tracking a change or restricting certain users from performing a change at certain period of time. We showed the interviewees some examples of bug reports in which the priority has changed along with related information such as the time, who did the change, and relevant comments about the priority changes. The exhibited examples set the context and scope for our discussion with the participants.

Then, we asked them to tell us some real-world situations when they needed to carefully check a change in bugs priority. These steps helped stimulate the practitioners' memories as well. As a next step, we asked them to think about an exhaustive set of reasons to change bugs priority after describing their experiences in changing, finding, and documenting these bugs priority so that it does not slow down addressing critical bugs while respecting deadlines and dealing with available resources. Based on these unstructured interviews, we built an initial 3-W model for bugs priority changes. We validated the initial model later with more participants via surveys and a quantitative validation using data collected from open-source systems.

[1]https://www.atlassian.com/software/jira
[2]https://www.bugzilla.org

5.2.3 Phase 2: Survey about the 3-W Model for Bugs Priority Changes

After deriving a bugs priority change model from the interviews, we designed a survey to understand their experiences and opinions about changes in bug's priority. The survey also aimed to elicit the individual roles who are responsible for changing bug's priority and when they make these changes along with the reasons that lead to increasing or decreasing the priority.

We distributed our 12-question survey to multiple software engineering groups, software testing groups, and software maintenance groups on several social media channels and a list of private emails of researchers with related background in software engineering and testing. We used the snowball sampling of the interviews for our survey by reaching out to our industry partners and asking them to advertise it to their contacts.

Table 5.3 shows a list of questions of our survey which was was carried out using Qualtrics [3]. The table shows the connections between our research questions and the questions of the survey. In the first section, we prepared three questions to capture demographic information about the background of the participants (number of years in the field, the level of education, and their current role/occupation).

Our intention in the second section of the survey is to gather information about the rationale of changing the priority of the bugs reports. So we asked our participants to choose from a list of possible reasons collected from the interviews from Phase 1 or propose a new reason to be added to our findings. Furthermore, we asked the participants whether they leave a comment about the rationale when they themselves change bug's priority and whether they change other attributes/fields in bugs reports while they are changing the priority. The goal of this question is to identify a possible correlation between bug's priority and other attributes, as well as to discover other reasons of changing bug's priority. Finally, we asked the participants whether they

[3]https://www.qualtrics.com

consider the priority to rank their list of tasks and organize their schedules around it.

The third section of the survey focused on when bugs priority are changed comparing to other temporal events such as the date of bug's creation, the data of bug's resolution, the date on which developers are assigned to the bug, and the date of the deployment or releasing new version of the software. Also, we asked our participants if they follow an approval or voting process or any other mechanisms prior to changing the priority of the bug.

In the last section, we asked the participants whether they change the priority themselves and how often they change it. We were also interested to know the group of people who changes the priority including their profiles, current occupations, and their role in the project. Furthermore, we asked the frequency of changing a bug's priority in general.

We discarded three responses based on the short time that they spent to take the survey (less than 5 minutes). We considered 38 survey responses (after discarding three responses), the participants took between 8 to 12 minutes to finish the survey.

5.2.4 Phase 3: Quantitative Analysis

Figure 5.4 shows the list of open-source systems that we analyzed in this study. The table shows the number of bugs and comments in bug reports for each project. We used Bugzilla API to fetch all bugs reports and the list of related comments along with changes on any attributes of the bug reports. Furthermore, we have collected all the commits belonging to all the considered project on their GitHub repositories. Overall, we have collected a total of $225,534$ bug reports belonging to 24 projects. Those bugs reports included more than 1.79 million changes in bugs reports and 1.35 million developers' comments related to these changes. In addition, these projects have $302,760$ GitHub commits with $15,000$ releases.

A bug report from Bugzilla is composed of several attributes listed below.

- *Creator:* The login name of the person who filed this bug (the reporter).
- *Creation time:* When the bug was created.
- *Keywords:* Each keyword mentioned on this bug.
- *Severity:* The current severity of the bug.
- *Resolution:* The current resolution of the bug, or an empty string if the bug is open.
- *Summary:* A summary of this bug.
- *Status:* The current status of the bug.
- *Priority:* The current priority of the bug.
- *Assignee:* The login name of the user to whom the bug is assigned.

We collected every change that happened on any attribute on the bug reports and extracted the following fields corresponding to each change.

- *Field Name:* The field that is changed.
- *Time:* The date-time stamp on when the change happened.
- *Old Value:* The value of the field before it has been changed.
- *New Value:* The value of the field after it has been changed.
- *Person:* The login name of the user who changed the value of the field.

The collected data is used to answer our research questions and find potential gaps between the developers' perception to changes in bugs priority and actual ones observed in the practice.

5.3 Evaluation

In this section, we answer the research questions by combining the results of the survey and the outcomes of the quantitative analysis on several open-source projects. Figure 5.2 represents the demographic information of the 38 participants of the survey. Their current occupation ranges between researcher, senior and junior software engineers, and QA/testers.

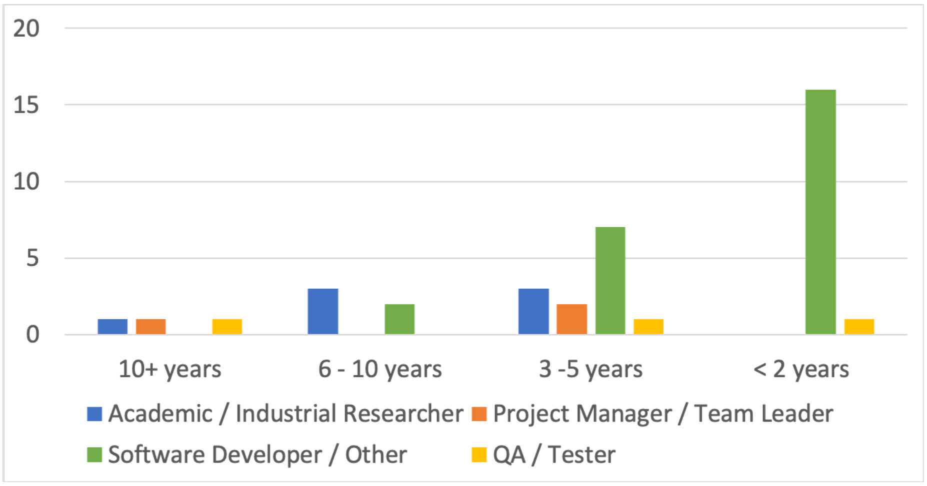


Figure 5.2: Current occupation and years of experience

5.3.1 RQ1: Why does the priority of a bug change?

Figure 5.3 shows that the majority of our participants agree that they consider bug's priority to manage their workload. Eight participants mentioned that they use the priority sometimes, and only 2 participants highlighted that they rarely look at the priority to manage their tasks. None of the participants expressed that they never consider the bug's priority. Our quantitative analysis shows that 100% of collected bugs have a priority assigned to them which proves the importance of bug's priority in managing bugs reports.

Figure 5.4 shows that only 2 participants said that they never changed bug's priority where the majority of them agree to the fact that they do need to change the bug's priority frequently. Furthermore, Figure 5.5 shows that most of the developers agree to change the priority once, twice, or more in general. Our quantitative analysis shows that more than 10% of collected bugs reports have their priority changed at least one time, and over 86% of those bugs had their priority changed at least two times.

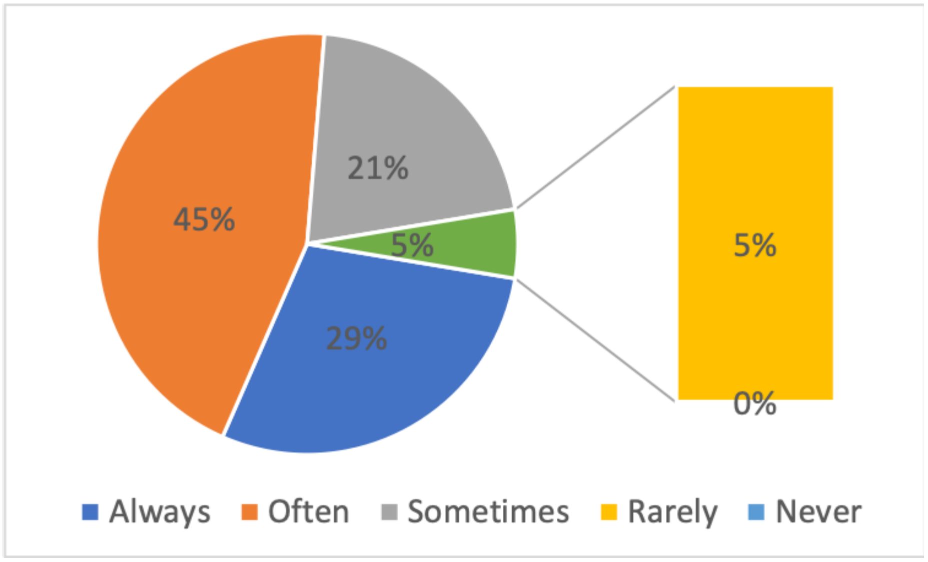


Figure 5.3: Survey Question: Do you consider bug's priority to manage your workload?

Figure 5.6 show the distribution of the votes among several possible reasons of changing bug's priority. It is clear that the dependencies between the bugs has high impact on changing the priority. From the response, we observe that the dependencies between the bugs is the biggest reason for changing bug's priority. Lack of time and high workload has the next biggest impact on priority change with 19 and 17 votes, respectively. There are 13 participants said that the initial priority value is always not accurate and hence they have to correct it to match the reality of the issue. We also found that 10 participants agree that the domain of project or the category of the bug report may affect the clarity of the bug report which makes a big difference whether changing the priority is needed or not. Other participants confirmed that changing priority by mistake does happen but it is not frequent.

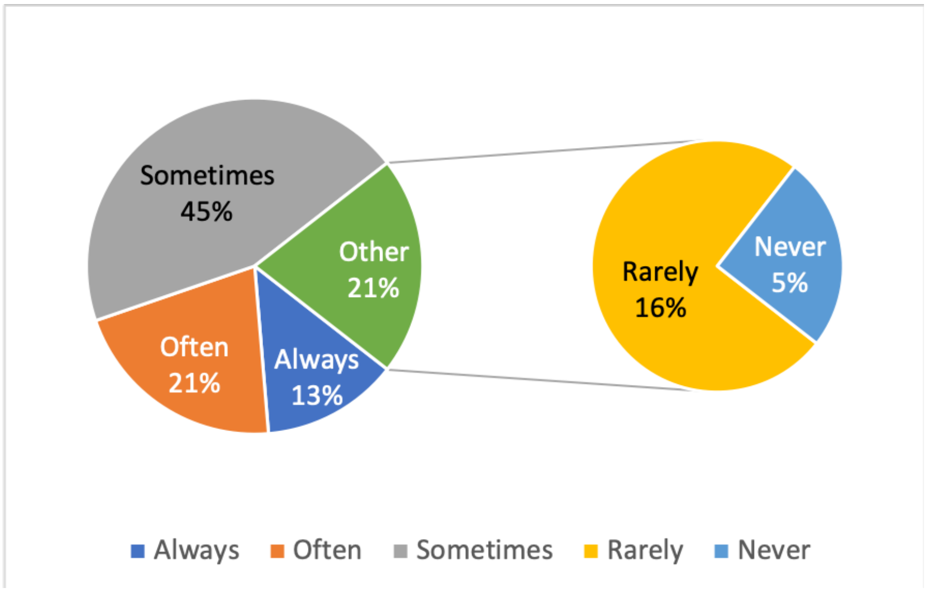


Figure 5.4: Survey Question: Do you change the priority of bugs?

Our quantitative analysis shows that 14% of the bug reports with priority changes have their priority changed twice from high to low and then from low to high. This outcome shows that bug's assignee may change the priority once it gets assigned to them to reduce the perceived urgency of the bug and to delay the delivery of the solution to the end-users. While it may not be among the best practices to change priorities to reduce the workload, the quantitative analysis of the bug reports show that this aspect is common when bugs priority are lowered.

We found that developers may change the priority based on their present workload in order to avoid any interruptions in the current sprint or simply due to lack of time. In fact, we looked into developers' comments in the collected bug reports and we found some comments complaining about workload or lack of time. We also found comments mentioning dependency with other bugs/issues from other teams/projects. Examples #1, #3, and #7 in Table 5.5 show the details of those comments.

Another finding is that developers might not see the necessity of assigned high priority or they might see the urgency in the assigned high priority of a bug. Examples #4 and #8 of Table 5.5 show that developers change the priority to better match the

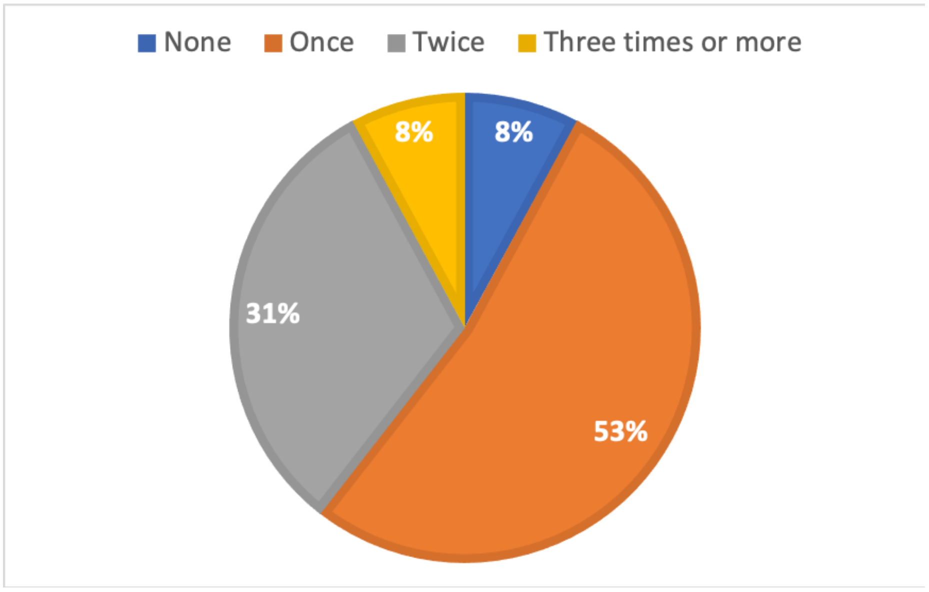


Figure 5.5: Survey Question: How many times do you change, in general, the priority per bug?

reality.

We also noticed that when the initial bug's description is not clear or not detailed enough, developers tend to introduce priority changes after some investigation based on the analysis of the comments in bug reports. We found also that there are some projects that do not have any priority changes, the reasons could be the lack of using the bug's priority to prioritize their workload or the bug's description is clear enough to make an accurate estimation of the priority of the bug.

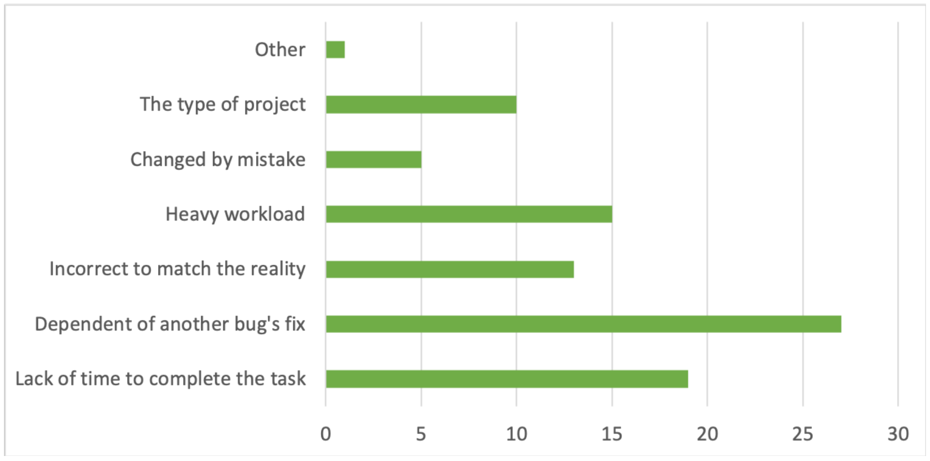


Figure 5.6: Survey Question: Why do you change bug's priority?

Regarding the documentation of bugs priority changes, most of the participants in our survey confirmed that they tend to leave a comment explaining the change; only nine responses said that they rarely or never document these changes. However, we found that developers are not following specific guidelines in documenting these changes and they are documented in an informal way.

We also noticed that description, status, and severity attributes are the most common attributes that get changed upon changing the priority. Based on participants' responses, only five of them mentioned that they have to pass through an approval process before changing the priority. The remaining participants agree that there is no process that restrict some developers from changing the priority at any time and for any bug report. The lack of this process to check and validate priority changes can be a reason for some suspicious changes due to high workload, release deadlines, showing less impact of a bug created by a developer, etc.

We found in our quantitative analysis that status, resolution, assignee, and target milestone are the fields that get changed when priority attribute is changed. In 27% of priority changes activities, we noticed that the assignee field changed. It may indicates that assigning the right developer to fix the bug is important to get the right estimation of the priority. We also discovered that 37% of priority change activities where priority changed at the same time with status and target milestone fields. We consider those activities as suspicious because they tend to delay the deliverable of the project by lowering the priority because there is an upcoming release deadline. To confirm this observation, we investigated and found that 44% of these bug reports get back to the same level of priority after a short period of time that can be associated with a release deadline.

By exploring some comments among developers in several of the bug reports in open-source projects, we found that many discussions are about raising or lowering priority or simply asking for clarity about the bug's description itself. This observation may confirm that there are some discussions happening before changing the priority between developers which can contribute to better explaining the reasons behind the changes. On the other hand, we found, by looking at the comments of developers, that bug's priority gets changed by accident as described in example #5 of Table 5.5. These accidental changes confirms again the importance of adding a mechanism in the pipeline of localizing and fixing bugs to validate the priority changes.

Key findings: The priority changes for the following reasons:

- The dependency of another bug's fix
- Incorrect priority
- Type / Domain of project
- Category of the bug report
- Lack of time / Heavy workload / Tight schedule
- Accident
- Hot-fix request
- Business requirements

Key findings: Bugs tracking systems track the changes of the bug's priority but they lack the ability to document the reasons to do such a change.

5.3.2 RQ2: Who does change priority of a bug?

Figure 5.7 shows that most of priority changes are carried out by developers, team leaders, or project managers. Out of 38, 31 responses show that developers change the bug's priority making them playing the biggest role in the change. It is not surprising because they are the ones who work on localizing and fixing the bugs. Other responses, 22 and 26, suggest that team leaders or project managers/owners could also change the priority to rush certain software features or meet future expectations or milestones.

In the quantitative analysis, we found that 28% of bugs reports with priority changes have been changed by their assignee. In addition, there is 19% of bug reports with priority changes where the priority is changed by their reporter or creator of the bug. We assume that the rest of the priority changes were performed by project leader, business analyst, or another developer. The lack of information to describe the role or the profile of each of the team members is also an issue in the open-source software and bug tracking systems. We note that there is no mechanism or approval process by which project's stakeholders can request the change and apply the change to the bug report.

🔑 **Key findings:** Most priority changes get changed by various project's stakeholders starting from developers to team leaders and project managers.

🔑 **Key findings:** Bug tracking systems track the individuals who make the changes on the priority but they lack the ability to:

- 💎 Restrict certain individuals to change the priority since they may not have the required knowledge and expertise of the addressed bug.
- 💎 Capture profile or role information about project's stakeholders.

5.3.3 RQ3: When does priority of a bug change?

Figure 5.8 shows that most of priority changes happen between the date in which bug gets assigned to the developer and date before releasing a new version of the software. Some answers claim that it is necessary to change the priority by the project manager who is assigning them because each developer has their own tasks

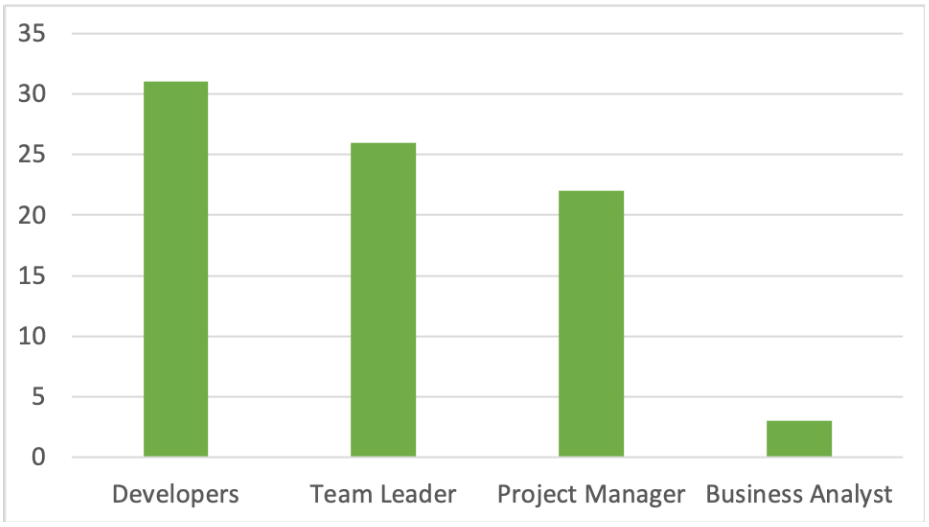


Figure 5.7: Survey Question: Who does change priority of a bug?

and therefore the priority should be tuned based on the type and the number of tasks assigned to the developer. Other explanation comes from best practices of agile methodology where they are advised to change the priority after scrum planning sessions with the development and business teams.

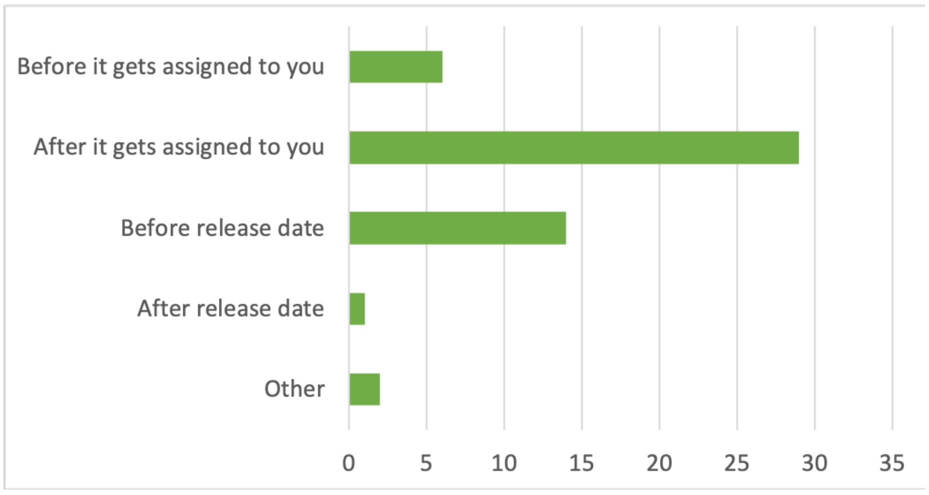


Figure 5.8: When do you change, in general, the priority of bugs?

In the quantitative analysis on bug reports from open-source projects, we found that the a bug's priority changes 2 times on an average with minimum of one time and maximum of 18 times. Interestingly, we found that a bug's priority changes in a relatively short period of time with an average 15 days from releasing a new version of the software. More surprisingly, we identified that there are about 44% of the bugs where the priority has been changed twice—the first change is to lower the priority and the second one is to reset it to the original priority of the bug. Thus, the bug's priority changes twice, once after developer or assignee reviews it, another time comes in a short period after the release date. This bad practice should be avoided since developers may tend to ship their code quickly with bugs in that some of them could be critical.

🔑 **Key findings:** Most priority changes happen between the date when the bug gets assigned to the developer and the date just before releasing new version of the software.

🔑 **Key findings:** Bug tracking systems track the timestamp when changes have happened but they lack the ability to:

- 💎 Lock the priority so that it cannot be changed after certain time or restrict certain individuals to change the priority after it passes a period of time or when it reaches certain status.
- 💎 Capture project's milestones along with their due date.

To summarize, we collected all our findings in the survey and the results of quantitative analysis. The outcome is presented in Table 5.6 as 3-W model to show the consolidated findings from both the survey and the quantitative analysis. We also present a list of recommendations for any future improvements by industrial and research communities. Therefore, we classified our findings and recommendations into three different groups to answer each of our research questions.

The first group is designed to answer the first research question as to *why does the priority change*. We found that the dependency of another bug could be one of the reasons besides some other obstacles that programmers encounter such as lack of time, heavy workload, or tight schedule. Also, we identified some cases where developers set incorrect priority for bug reports in which they are not knowledgeable enough to do so. Another reasons could be related to business requirements where they have to prioritize some bugs over other bugs considering the severity of the bugs. Likewise, the priority may need to be changed if the bug report is considered to be a hot-fix request and thus it needs to be tackled by the development team right away. According to our quantitative analysis, we noticed that some projects do not have any changes in priority due to the clarity of the description of the bug report, the size of the project, or the size of the development team. Similarly, we found that the changes in the priority of some bug reports are different than the priority of other bug reports in the same project due to the category of the bug report such as security, functionality, or user interface related bugs. Lastly, we noticed some cases where the developers changed the priority by accident. As a recommendation, due to lack of documentation upon changing the priority, we recommend that all bug tracking systems should have the ability to track the changes in priority with an appropriate documentation noted by the user who makes the change. Capturing such information will help in priority predication process to accurately predicate the priority of bug reports and therefore improve the bug triage process. Also, we recommend adapting a standard in bug report documentation to avoid an ambiguity in the process.

The next group is focused to answer the second research question as to *who changes the priority*. According to our findings from both the survey and the quantitative analysis, we discovered that there is no rule on who is allowed or not allowed to change a priority. We encounter several cases where developers, team leaders, or project owners have the permission to change the priority without prior approval process or team decision making. As a recommendation, we suggest that bug tracking software should have the ability to prevent some users from changing the priority and doing so by knowing more about the team structure or hierarchy.

The last group is to answer the third research question as to *when does a priority change*. We discovered that a priority gets changed anytime as long as the bug has not been resolved. We have seen examples where it gets changed after the creation, before assigning to the developer, after assigning to the developer, before or upon closing the bug report. More importantly, we noticed that the priorities get changed in the last phase of bug resolution—a short period of time just before closing the bug report and set a resolution for it. We recommend that bug tracking systems should be aware of project's future releases, milestones, and the current bugs and features pipeline targeted to the release. Subsequently, the bug tracking software should be able to restrict certain users from changing the priority in a critical time to prevent any delays in resolving the bug reports.

5.4 Threats to Validity

We want to acknowledge several threats to the validity of the paper such as the factors that can bias our empirical study. These factors can be classified into three categories: internal validity, construct internal, and external validity. Construct validity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity threats are related to the absence of similar work that uses bug localization technique to generate a dependency graph among several bug reports and therefore recommend those bugs in sequential order. For that reason, we compared our proposal with different mono-objective formulations that use one metric only like the score of bug priority. The developers were asked to evaluate different systems using different tools. We did not allow developers to evaluate different tools on the same system. The developers were distributed among the systems and tools based on their background/expertise to ensure almost the same level for all systems and tools. When each developer is asked to evaluate one different tool per system, we reduce the potential bias in the experiments since they are using the tools for the first time and they are exploring each time a new system. Our results show that the productivity has gotten better for the majority of our developers regardless of their experience and skills set.

External validity refers to the fact that our survey has been conducted by 29 developers with a variety of skills and number of experience. Thus, we can affirm that our results will hold its accuracy with a different set of developers with different level of expertise or knowledge. Also, time collection was left to each individual developer who manually noted the time they started and finished localizing a defect. This could have resulted in introducing error as every developer performed differently.

Finally, External validity could be related to the type of projects we used in the survey in which we used six different widely-used open-source systems belonging to the different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, other programming languages, and to other practitioners.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error, which is commonly used in the SBSE community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance.

5.5 Conclusions

In this thesis, we used a combination of interviews, a survey, and bug reports analysis to understand the changes in bugs priority and their rationale. We started first with a set of interviews with practitioners to define a bugs priority change model. Then, we performed a large online survey to gather the experiences of practitioners with the rationale, frequency, and experiences of changing the priority of bugs. We have also collected a large data-set of bugs priority change on open-source projects. We looked into actions that happen in bug reports such as the date when priority changes happen, the reasons beyond changing the priority documented in the comments, and the profile of the users who changed the priority. The quantitative validation on this created data-set revealed several areas of improvements as discussed in the implications section.

The outcomes of this empirical study can be used to build tools for automatically validating the changes' request of bugs priority. We are planning as part of our future work to leverage machine learning to check and validate the bugs priority changes submitted by developers based on the data-set collected in this study *Almhana and Kessentini* (2020c).

Study	**Description / Technique Used**	**Have they addressed priority changes?**
Yang et al. *Yang et al.* (2014)	Extract and identify multi-feature (e.g., Component, product, priority and severity) from bug report	No
Tian et al. *Tian et al.* (2013, 2015)	Use several factors such as temporal, textual, author, related-report, severity, and product, to predict the priority level of a bug report	No
Sharma et al. *Sharma et al.* (2012)	Use Support Vector Machine, Naive Bayes, K-Nearest Neighbors and Neural Network in predicting the priority of bugs	No
Kanwal et al. *Kanwal and Maqbool* (2010, 2012)	Propose a priority recommendation module based on Naïve Bayes and Support Vector Machine.	No
Yu et al. *Yu et al.* (2010)	Utilize neural network techniques to predict the priorities of bugs	No
Alenezi et al. *Alenezi and Banitaan* (2013)	Present an approach to use different machine learning algorithms namely Naive Bayes, Decision Trees, and Random Forest	No
Kumari et al. *Kumari and Singh* (2018)	Build classifiers using machine learning and Naïve Bayes and Deep Learning techniques	No

Table 5.2: Summary of previous studies about bug priority predictions.

Question#	**Survey question**	**Research Question / Intention**
1	What is your highest level of education related to Computer Science?	Background Information
2	What is your current occupation?	
3	How many years have you worked with software?	
4	Why do you change bug's priority?	Why does the priority of the bug change?
5	Do you leave a comment about the rationale when you change bug's priority?	
6	What are the attributes that you change when you change bug's priority?	
7	Do you consider bug's priority to manage your workload?	
8	When do you change, in general, the priority of bugs?	When does the priority of the bug change?
9	Is there any approval process or poll/vote mechanism to check your change of bug's priority?	
10	Who does change bug's priority?	Who does change the priority of the bug?
11	Do you change the priority of bugs?	
12	How many times do you change, in general, the priority per bug?	

Table 5.3: The traceability between our research questions and survey questions

Project Name	#Bugs	#Comments	#Commits	#Versions
Eclipse Plat-form	118309	761180	8190	5201
BIRT	23270	111178	41290	437
AspectJ	3049	16666	46910	112
JDT	59780	367172	24222	5723
Buildship	482	2304	3314	37
JGit	1351	7254	7655	174
openj9	9	36	41948	42
PDT	6062	31580	9490	487
TCF	1238	5896	4830	28
SW360	2	5	512	12
Antenna	2	4	773	28
Hawkbit	2	2	2267	21
Californiur	61	212	1927	32
Kapua	4	13	4078	24
GEF	3167	14612	5113	7
Ditto	2	2	4343	19
Vorto	160	467	1987	31
Titan	563	1857	8237	15
Jetty	3813	16661	66427	340
BPEL	386	1341	1047	9
e4	3788	21057	993	1122
Milo	1	1	759	24
Che	31	96	8372	163
OMR	2	12	8076	1

Table 5.4: List of studied projects along with the number of bugs, comments, Github commits, and releases

Example#	**Bug ID**	**Project**	**Comment**
1	150807, 151061	JDT	Downgrading priority since we will probably not have time for this.
2	33897, 34076, 35075	Eclipse Platform	There are no plans for the UI team to work on this defect until higher priority items are addressed.
3	217891, 233481	JDT	Ownership has changed for the javadoc comments bugs, but I surely will not have enough time to fix your bug during the 3.5 development process,
4	151612, 170140	Eclipse Platform	Lowering priority to better match reality.
5	75829	Eclipse Platform	My apology, I inadvertently changed the priority when I changed the severity, I'm changing it back now.
6	50888, 52115	JDT	Resetting priority to P3. Will be reassessed for the next release.
7	191927	BIRT	Firefox hasn't fix this bug yet. Set the priority to p5.
8	21652	Eclipse Platform	Lowering priority to P2 (P1 means that this is a "stop-ship" bug report)

Table 5.5: Developers' comments from several open source software regarding the changes on the priority of the bug.

Research Question	**Findings**	**Recommendations**
Why does the priority of a bug change?	💎 The dependency of another bug's fix 💎 Incorrect priority 💎 Heavy workload /Tight schedule 💎 Category of bug report 💎 Hot-fix request 💎 Business requirements 💎 Type of project 💎 On accident	🚀 Bug tracking systems should have the ability to document the reasons of changing priority 🚀 Priority predication systems should rely on priority changes to improve their predication model 🚀 Standardize documentation methods based on different projects' domains and bugs' categories
Who does change priority of a bug?	💎 Stakeholders including developers, team leaders, project owners	🚀 Bug tracking systems should have the ability to restrict certain users from changing the priority if they don't have permissions to do so. 🚀 Bugs tracking systems should be aware of team structure and the role of each stakeholder.
When does priority of a bug change?	💎 Priority changes happen between the date the bug gets assigned to the developer and date before releasing new version of the software.	🚀 Bugs tracking systems should prevent stakeholders from changing the priority unnecessarily if the bug milestone is close to the release date or if the bug in active or pending status. 🚀 Bugs tracking systems should be aware of project's milestones and timelines.

Table 5.6: 3-W Model Findings and Recommendations

CHAPTER VI

Conclusion

6.1 Summary and Future Work

The main contributions of the proposed work can be summarized as follows:

- We propose an automated approach to localize and rank potential relevant methods for bug reports as an extension of our previous work limited to class level recommendations. Our approach finds a trade-off between minimizing the number of recommended methods and maximizing the correctness of the proposed solution using a hybrid multi-objective algorithm. The correctness of the recommended methods is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. Our approach uses the main steps, the first step finds the best set of classes satisfying the two conflicting criteria of relevance and number of classes to recommend using a global search based on NSGA-II. The second step is to locate the most appropriate methods to inspect, using a local multi-objective search based on Simulated Annealing (MOSA) from the list of classes identified in the first step.

- We propose an approach for bugs management by taking into consideration both the severity and dependencies between reports. Our solution is based on the use of multi-objective search to find a trade-off between these two conflicting objectives. The validation of our work shows that there were significant time savings when developers inspected bugs comparing to existing methods treating each bug individually as first come first serve or relaying on priority scores only.

- we use a combination of interviews, a survey, and bug reports analysis to understand the changes in bugs priority and their rationale. We started first with a set of interviews with practitioners to define a bugs priority change model. Then, we performed a large online survey to gather the experiences of practitioners with the rationale, frequency, and experiences of changing the priority of bugs.

As part of our future work, we envision the extension of this approach to improve the bugs management process by recommending developers to be assigned for bugs based on their background and prior expertise. The users can interact more with the suggested recommendations in order to update the assignments. On the other hand, we are planning as part of our future work to leverage machine learning to check and validate the bugs priority changes submitted by developers.

6.2 Publications List

- Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, Ali Ouni: Recommending relevant classes for bug reports using multi-objective search. ASE 2016 conference, acceptance rate 16% : 286-295, IEEE

- Almhana, R., Ferreira, T., Kessentini, M. and Sharma, T., 2020, September. Understanding and Characterizing Changes in Bugs Priority: The Practitioners' Perceptive. In 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 87-97), acceptance rate 24%. IEEE

- Almhana, R., Kessentini, M. and Mkaouer, W., 2020. Method-Level Bug Localization Using Hybrid Multi-objective Search. Information and Software Technology Journal, Volume 131, 32 pages, Elsevier, Impact Factor 2.73
- Almhana, R. and Kessentini, M., 2020. Detecting Dependencies Between Bug Reports to Improve Bugs Triage. Automated Software Engineering Journal, 26 pages, to appear, Impact Factor 1.97

BIBLIOGRAPHY

Abdelmoez, W., M. Kholief, and F. M. Elsalmy (2012), Bug fix-time prediction model using naïve bayes classifier, in *2012 22nd International Conference on Computer Theory and Applications (ICCTA)*, pp. 167–172, IEEE.

Alenezi, M., and S. Banitaan (2013), Bug reports prioritization: Which features and classifier to use?, in *2013 12th International Conference on Machine Learning and Applications*, vol. 2, pp. 112–116, IEEE.

Alizadeh, V., M. Kessentini, M. W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai (2018), An interactive and dynamic search-based approach to software refactoring recommendations, *IEEE Transactions on Software Engineering*, *46*(9), 932–961.

Almhana, R., and M. Kessentini (2020a), Bug reports data, http://bit.ly/2NUyion, accessed: 2020-01-20.

Almhana, R., and M. Kessentini (2020b), Methods level data for bugs localization, http://www-personal.umd.umich.edu/ marouane/tsedata.zip, accessed: 2018-10-01.

Almhana, R., and M. Kessentini (2020c), Replication package, uRL: https://sites.google.com/view/ ase2020-bugs-priority.

Almhana, R., W. Mkaouer, M. Kessentini, and A. Ouni (2016), Recommending relevant classes for bug reports using multi-objective search, in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 286–295, ACM.

AlOmar, E. A., M. W. Mkaouer, A. Ouni, and M. Kessentini (2019), On the impact of refactoring on the relationship between quality attributes and design metrics, in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11, IEEE.

Aman, H., S. Amasaki, T. Yokogawa, and M. Kawahara (2019), Empirical study of abnormality in local variables and its application to fault-prone java method analysis, *Journal of Software: Evolution and Process*, p. e2220.

Ashok, B., J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala (2009), Debugadvisor: a recommender system for debugging, in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM*

SIGSOFT symposium on The foundations of software engineering, pp. 373–382, ACM.

Behl, D., S. Handa, and A. Arora (2014), A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf, in *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pp. 294–299, IEEE.

Bettenburg, N., R. Premraj, T. Zimmermann, and S. Kim (2008), Duplicate bug reports considered harmful… really?, in *Software maintenance, 2008. ICSM 2008. IEEE international conference on*, pp. 337–345, IEEE.

Blei, D. M., A. Y. Ng, and M. I. Jordan (2003), Latent dirichlet allocation, *Journal of machine Learning research*, *3*(Jan), 993–1022.

Bruegge, B., and A. H. Dutoit (2004), *Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)*, vol. 2004, Prentice Hall.

Canfora, G., M. Ceccarelli, L. Cerulo, and M. Di Penta (2011), How long does a bug survive? an empirical study, in *2011 18th Working Conference on Reverse Engineering*, pp. 191–200, IEEE.

Chaturvedi, K., and V. Singh (2012), Determining bug severity using machine learning techniques, in *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pp. 1–6, IEEE.

Chung, L., B. Nixon, E. Yu, and J. Mylopoulos (2000), Non-functional requirements, *Software Engineering*.

Codoban, M., S. S. Ragavan, D. Dig, and B. Bailey (2015), Software history under the lens: A study on why and how developers examine it, in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–10, IEEE.

Czyzżak, P., and A. Jaszkiewicz (1998), Pareto simulated annealing—a metaheuristic technique for multiple-objective combinatorial optimization, *Journal of Multi-Criteria Decision Analysis*, *7*(1), 34–47.

Deb, K., and S. Gupta (2011), Understanding knee points in bicriteria problems and their implications as preferred solution principles, *Engineering optimization*, *43*(11), 1175–1204.

Deb, K., A. Pratap, S. Agarwal, and T. Meyarivan (2002), A fast and elitist multiobjective genetic algorithm: Nsga-ii, *IEEE transactions on evolutionary computation*, *6*(2), 182–197.

Dommati, S. J., R. Agrawal, S. S. Kamath, et al. (2013), Bug classification: Feature extraction and comparison of event model using na\" ive bayes approach, *arXiv preprint arXiv:1304.1677*.

Dreyton, D., A. A. Araújo, A. Dantas, Á. Freitas, and J. Souza (2015), Search-based bug report prioritization for kate editor bugs repository, in *International Symposium on Search Based Software Engineering*, pp. 295–300, Springer.

Dreyton, D., A. A. Araújo, A. Dantas, R. Saraiva, and J. Souza (2016), A multi-objective approach to prioritize and recommend bugs in open source repositories, in *International Symposium on Search Based Software Engineering*, pp. 143–158, Springer.

Dumais, S. T. (2004), Latent semantic analysis, *Annual review of information science and technology*, *38*(1), 188–230.

Easterbrook, S., J. Singer, M.-A. Storey, and D. Damian (2008), Selecting empirical methods for software engineering research, in *Guide to advanced empirical software engineering*, pp. 285–311, Springer.

Emmerich, M. T., and A. H. Deutz (2018), A tutorial on multiobjective optimization: fundamentals and evolutionary methods, *Natural computing*, *17*(3), 585–609.

Enslen, E., E. Hill, L. Pollock, and K. Vijay-Shanker (2009), Mining source code to automatically split identifiers for software analysis, in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pp. 71–80, IEEE.

Fischer, M., M. Pinzger, and H. Gall (2003), Analyzing and relating bug report data for feature tracking, in *WCRE*, vol. 3, p. 90.

Geng, J., S. Ying, X. Jia, T. Zhang, X. Liu, L. Guo, and J. Xuan (2018), Supporting many-objective software requirements decision: An exploratory study on the next release problem, *IEEE Access*, *6*, 60,547–60,558.

Ghannem, A., M. Kessentini, and G. El Boussaidi (2011), Detecting model refactoring opportunities using heuristic search, in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 175–187.

Ghannem, A., G. El Boussaidi, and M. Kessentini (2014), Model refactoring using examples: a search-based approach, *Journal of Software: Evolution and Process*, *26*(7), 692–713.

Goyal, N., N. Aggarwal, and M. Dutta (2015), A novel way of assigning software bug priority using supervised classification on clustered bugs data, in *Advances in Intelligent Informatics*, pp. 493–501, Springer.

Guo, P. J., T. Zimmermann, N. Nagappan, and B. Murphy (2010), Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows, in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 495–504.

Harman, M., and B. F. Jones (2001), Search-based software engineering, *Information and software Technology*, *43*(14), 833–839.

Harman, M., S. A. Mansouri, and Y. Zhang (2012), Search-based software engineering: Trends, techniques and applications, *ACM Computing Surveys (CSUR)*, *45*(1), 11.

Henard, C., M. Papadakis, and Y. Le Traon (2014), Mutation-based generation of software product line test configurations, in *International Symposium on Search Based Software Engineering*, pp. 92–106, Springer.

Hilton, M., N. Nelson, T. Tunnell, D. Marinov, and D. Dig (2017), Trade-offs in continuous integration: assurance, security, and flexibility, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 197–207.

Huang, Q., D. Lo, X. Xia, Q. Wang, and S. Li (2017), Which packages would be affected by this bug report?, in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 124–135, IEEE.

Jones, J. A. (2008), Semi-automatic fault localization, Ph.D. thesis, Georgia Institute of Technology.

Kanwal, J., and O. Maqbool (2010), Managing open bug repositories through bug report prioritization using svms, in *Proceedings of the International Conference on Open-Source Systems and Technologies, Lahore, Pakistan*, pp. 22–24.

Kanwal, J., and O. Maqbool (2012), Bug prioritization to facilitate bug report triage, *Journal of Computer Science and Technology*, *27*(2), 397–412.

Keller, A. A. (2019), *Multi-Objective Optimization in Theory and Practice II: Metaheuristic Algorithms*, Bentham Science Publishers.

Kessentini, M., and A. Ouni (2017), Detecting android smells using multi-objective genetic programming, in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 122–132, IEEE.

Kessentini, M., A. Bouchoucha, H. Sahraoui, and M. Boukadoum (2010), Example-based sequence diagrams to colored petri nets transformation using heuristic search, in *European Conference on Modelling Foundations and Applications*, pp. 156–172, Springer, Berlin, Heidelberg.

Kessentini, M., P. Langer, and M. Wimmer (2013a), Searching models, modeling search: On the synergies of sbse and mde, in *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pp. 51–54, IEEE.

Kessentini, M., R. Mahaouachi, and K. Ghedira (2013b), What you like in design use to correct bad-smells, *Software Quality Journal*, *21*(4), 551–571.

Kumari, M., and V. Singh (2018), An improved classifier based on entropy and deep learning for bug priority prediction, in *International Conference on Intelligent Systems Design and Applications*, pp. 571–580, Springer.

Lam, A. N., A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen (2017), Bug localization with combination of deep learning and information retrieval, in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 218–229, IEEE.

Lamkanfi, A., S. Demeyer, E. Giger, and B. Goethals (2010), Predicting the severity of a reported bug, in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pp. 1–10, IEEE.

Lamkanfi, A., S. Demeyer, Q. D. Soetens, and T. Verdonck (2011), Comparing mining algorithms for predicting the severity of a reported bug, in *2011 15th European Conference on Software Maintenance and Reengineering*, pp. 249–258, IEEE.

Li, Z., L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai (2006), Have things changed now?: an empirical study of bug characteristics in modern open source software, in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 25–33, ACM.

Liblit, B., A. Aiken, A. X. Zheng, and M. I. Jordan (2003), Bug isolation via remote program sampling, in *ACM Sigplan Notices*, vol. 38, pp. 141–154, ACM.

Loyola, P., K. Gajananan, and F. Satoh (2018), Bug localization by learning to rank and represent bug inducing changes, in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pp. 657–665, ACM.

Lukins, S. K., N. A. Kraft, and L. H. Etzkorn (2010), Bug localization using latent dirichlet allocation, *Information and Software Technology*, *52*(9), 972–990.

Mansoor, U., M. Kessentini, P. Langer, M. Wimmer, S. Bechikh, and K. Deb (2015), Momm: Multi-objective model merging, *Journal of Systems and Software*, *103*, 423–439.

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

Núñez, A., M. G. Merayo, R. M. Hierons, and M. Núñez (2013), Using genetic algorithms to generate test sequences for complex timed systems, *Soft Computing*, *17*(2), 301–315.

Ramirez, A., J. R. Romero, and S. Ventura (2019), A survey of many-objective optimisation in search-based software engineering, *Journal of Systems and Software*, *149*, 382–395.

Rao, S., and A. Kak (2011), Retrieval from software libraries for bug localization: a comparative study of generic and composite text models, in *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 43–52, ACM.

Saha, R. K., M. Lease, S. Khurshid, and D. E. Perry (2013), Improving bug localization using structured information retrieval, in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 345–355, IEEE.

Saha, R. K., J. Lawall, S. Khurshid, and D. E. Perry (2014), On the effectiveness of information retrieval based bug localization for c programs, in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pp. 161–170, IEEE.

Salton, G., A. Wong, and C.-S. Yang (1975), A vector space model for automatic indexing, *Communications of the ACM*, *18*(11), 613–620.

Sharma, M., P. Bedi, K. Chaturvedi, and V. Singh (2012), Predicting the priority of a reported bug using machine learning techniques and cross project validation, in *2012 12th International Conference on Intelligent Systems Design and Applications (ISDA)*, pp. 539–545, IEEE.

Shelburg, J., M. Kessentini, and D. R. Tauritz (2013), Regression testing for model transformations: A multi-objective approach, in *International Symposium on Search Based Software Engineering*, pp. 209–223, Springer.

Shihab, E., A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto (2013), Studying re-opened bugs in open source software, *Empirical Software Engineering*, *18*(5), 1005–1042.

Sun, C., D. Lo, X. Wang, J. Jiang, and S.-C. Khoo (2010), A discriminative model approach for accurate duplicate bug report retrieval, in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 45–54, ACM.

Tan, P.-N., et al. (2006), *Introduction to data mining*, Pearson Education India.

Tantithamthavorn, C., S. L. Abebe, A. E. Hassan, A. Ihara, and K. Matsumoto (2018), The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization, *Information and Software Technology*, *102*, 160–174.

Tao, Y., Y. Dang, T. Xie, D. Zhang, and S. Kim (2012), How do software engineers understand code changes? an exploratory study in industry, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11.

Thung, F., D. Lo, L. Jiang, F. Rahman, P. T. Devanbu, et al. (2012), When would this bug get reported?, in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 420–429, IEEE.

Tian, Y., D. Lo, and C. Sun (2012), Information retrieval based nearest neighbor classification for fine-grained bug severity prediction, in *2012 19th Working Conference on Reverse Engineering*, pp. 215–224, IEEE.

Tian, Y., D. Lo, and C. Sun (2013), Drone: Predicting priority of reported bugs by multi-factor analysis, in *2013 IEEE International Conference on Software Maintenance*, pp. 200–209, IEEE.

Tian, Y., D. Lo, X. Xia, and C. Sun (2015), Automated prediction of bug report priority using multi-factor analysis, *Empirical Software Engineering*, *20*(5), 1354–1383.

Uddin, J., R. Ghazali, M. M. Deris, R. Naseem, and H. Shah (2017), A survey on bug prioritization, *Artificial Intelligence Review*, *47*(2), 145–180.

Ulungu, E., J. Teghem, P. Fortemps, and D. Tuyttens (1999), Mosa method: a tool for solving multiobjective combinatorial optimization problems, *Journal of multicriteria decision analysis*, *8*(4), 221.

Valdivia Garcia, H., and E. Shihab (2014), Characterizing and predicting blocking bugs in open source projects, in *Proceedings of the 11th working conference on mining software repositories*, pp. 72–81, ACM.

Wang, S., and D. Lo (2014a), Version history, similar report, and structure: Putting them together for improved bug localization, in *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pp. 53–63, ACM, New York, NY, USA, doi:10.1145/2597008.2597148.

Wang, S., and D. Lo (2014b), Version history, similar report, and structure: Putting them together for improved bug localization, in *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 53–63, ACM.

Wen, M., R. Wu, and S.-C. Cheung (2016), Locus: Locating bugs from software changes, in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 262–273, IEEE.

Wilcoxon, F., S. Katti, and R. A. Wilcox (1970), Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test, *Selected tables in mathematical statistics*, *1*, 171–259.

Wong, C.-P., Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei (2014), Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis, in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pp. 181–190, IEEE.

Wong, W. E., R. Gao, Y. Li, R. Abreu, and F. Wotawa (2016), A survey on software fault localization, *IEEE Transactions on Software Engineering*, *42*(8), 707–740.

Xia, X., D. Lo, M. Wen, E. Shihab, and B. Zhou (2014), An empirical study of bug report field reassignment, in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 174–183, IEEE.

Xiao, Y., J. Keung, K. E. Bennin, and Q. Mi (2018), Machine translation-based bug localization technique for bridging lexical gap, *Information and Software Technology*, *99*, 58–61.

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

Yang, G., T. Zhang, and B. Lee (2014), Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports, in *2014 IEEE 38th Annual Computer Software and Applications Conference*, pp. 97–106, IEEE.

Ye, X., R. Bunescu, and C. Liu (2014), Learning to rank relevant files for bug reports using domain knowledge, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 689–699, ACM.

Ye, X., R. Bunescu, and C. Liu (2015), Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation, *IEEE Transactions on Software Engineering*, *42*(4), 379–402.

Ye, X., R. Bunescu, and C. Liu (2016a), Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation, *IEEE Transactions on Software Engineering*, *42*(4), 379–402.

Ye, X., H. Shen, X. Ma, R. Bunescu, and C. Liu (2016b), From word embeddings to document similarities for improved information retrieval in software engineering, in *Proceedings of the 38th international conference on software engineering*, pp. 404–415, ACM.

Youm, K. C., J. Ahn, and E. Lee (2017), Improved bug localization based on code change histories and bug reports, *Information and Software Technology*, *82*, 177–192.

Yu, L., W.-T. Tsai, W. Zhao, and F. Wu (2010), Predicting defect priority based on neural networks, in *International Conference on Advanced Data Mining and Applications*, pp. 356–367, Springer.

Zaman, S., B. Adams, and A. E. Hassan (2011), Security versus performance bugs: a case study on firefox, in *Proceedings of the 8th working conference on mining software repositories*, pp. 93–102.

Zaman, S., B. Adams, and A. E. Hassan (2012), A qualitative study on performance bugs, in *2012 9th IEEE working conference on mining software repositories (MSR)*, pp. 199–208, IEEE.

Zanetti, M. S., I. Scholtes, C. J. Tessone, and F. Schweitzer (2013), Categorizing bugs with social networks: a case study on four open source software communities, in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 1032–1041, IEEE Press.

Zhang, T., J. Chen, G. Yang, B. Lee, and X. Luo (2016), Towards more accurate severity prediction and fixer recommendation of software bugs, *Journal of Systems and Software*, *117*, 166–184.

Zheng, A. X., M. I. Jordan, B. Liblit, M. Naik, and A. Aiken (2006), Statistical debugging: simultaneous identification of multiple bugs, in *Proceedings of the 23rd international conference on Machine learning*, pp. 1105–1112, ACM.

Zhou, J., H. Zhang, and D. Lo (2012), Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports, in *Proceedings of the 34th International Conference on Software Engineering*, pp. 14–24, IEEE Press.

Zimmermann, T., R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss (2010), What makes a good bug report?, *IEEE Transactions on Software Engineering*, *36*(5), 618–643.

Zimmermann, T., N. Nagappan, P. J. Guo, and B. Murphy (2012), Characterizing and predicting which bugs get reopened, in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1074–1083, IEEE.

Zou, W., D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu (2018), How practitioners perceive automated bug report management techniques, *IEEE Transactions on Software Engineering.*