Pós-Graduação em Ciência da Computação

"A Bug Report Analysis and Search Tool"

By

Yguaratã Cerqueira Cavalcanti

M.Sc. Dissertation

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, JULY/2009

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

Yguaratã Cerqueira Cavalcanti

"A Bug Report Analysis and Search Tool"

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

A M.Sc. Dissertation presented to the Federal University of Pernambuco in partial fulfillment of the requirements for the degree of M.Sc. in Computer Science.

Advisor: *Silvio Romero de Lemos Meira*
Co-Advisor: *Eduardo Santana de Almeida*

RECIFE, JULY/2009

Cavalcanti, Yguaratã Cerqueira.
A bug report analysis and search tool / Yguaratã Cerqueira Cavalcanti. - Recife : O autor, 2009.
xiii, 109 folhas : il., fig., tab.

Dissertação (mestrado) - Universidade Federal de Pernambuco. CIN. Ciência da Computação, 2009.

Inclui bibliografia, glossário e apêndice.

1. Engenharia de software. 2.Gerenciamento de configuração de software. 3.Manutenção de software. I. Título.

005.1 CDD (22.ed.) MEI-2009-096

Dissertação de Mestrado apresentada por **Yguaratã Cerqueira Cavalcanti** Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título, **"A Bug Report Analysis and Search Tool"**, orientada pelo **Prof. Silvio Romero de Lemos Meira** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Alexandre Marcos Lins de Vasconcelos
Centro de Informática / UFPE

Prof. Uirá Kulesza
Departamento de Informática e Matemática Aplicada/UFRN

Prof. Silvio Romero de Lemos Meira
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 3 de julho de 2009.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

I dedicate this dissertation to myself and all my family, friends and professors who gave me all necessary support to get here.

Acknowledgements

I would like to thank and dedicate this dissertation to the following people:

My Familly My mother Luiza, my father Cícero, my brothers Tayguara and Nick, Adriana, João, my niece Isys

My friends from Maceió Hugo, Marcelo, Laudeci, Alberto, Fernando, Luis Antonio, Igor, Camila Sousa, Melina, Carla, Juliano, Eduardo Cardoso

My friends from UFPE Lucas Lima, Edurado Ribas, Edeilson, Paulo Moura, Daniel, Petrus, Mario Godoy, and Danuza

My friends from Recife Diego Cardozo, Thiago Xupeta, People from *Casa do Mar*, Geovane, Robson, Rogério Nibon, Manoel Moura, Maíra Rodrigues and her family, Maíra Gamarra, Daniel Queiroga, Vitor Hora

My adivisors Eduardo Almeida and Silvio Meira

All people from RiSE

I would like to especially thank Eduardo Almeida, who never gave up on our work and always helped me every step of this dissertation.

Finally, I am sorry if you think your name should be listed here and it is not, but I cannot remember all people I met during these years. Because of that, I think that the acknowledgements of a dissertation should be written/updated during all the period of a M.Sc./Ph.D (or any other course). There is a lot of people we met during this time that contributed very much to our growth (even with a little kind word) and should be listed here.

I open my eyes each morning I rise, to find a true thought, know that it's real, I'm lucky to breathe, I'm lucky to feel, I'm glad to wake up, I'm glad to be here, with all of this world, and all of it's pain, all of it's lies, and all of it's flipped down, I still feel a sense of freedom, so glad I'm around,

It's my freedom, can't take it from me, i know it, it won't change, but we need some understanding, I know we'll be all right.

—S.O.JA. (Open My Eyes)

Resumo

Manutenção e evolução de *software* são atividades caracterizadas pelo seu enorme custo e baixa velocidade de execução. Não obstante, elas são atividades inevitáveis para garantir a qualidade do *software* – quase todo *software* bem sucedido estimula os usuários a fazer pedidos de mudanças e melhorias. Sommerville é ainda mais enfático e diz que mudanças em projetos de *software* são um fato. Além disso, diferentes estudos têm afirmado ao longo dos anos que as atividades de manutenção e evolução de *software* são as mais caras do ciclo de desenvolvimento, sendo responsável por cerca de até 90% dos custos.

Todas essas peculiaridades da fase de manutenção e evolução de *software* leva o mundo acadêmico e industrial a investigar constantemente novas soluções para reduzir os custos dessas atividades. Neste contexto, Gerência de Configuração de Software (GCS) é um conjunto de atividades e normas para a gestão da evolução e manutenção de *software*; GCS define como são registradas e processadas todas as modificações, o impacto das mesmas em todo o sistema, dentre outros procedimentos. Para todas estas tarefas de GCM existem diferentes ferramentas de auxílio, tais como sistemas de controle de versão e *bug trackers*. No entanto, alguns problemas podem surgir devido ao uso das mesmas, como por exemplo o problema de atribuição automática de responsável por um *bug report* e o problema de duplicação de *bug reports*.

Neste sentido, esta dissertação investiga o problema de duplicação de *bug reports* resultante da utilização de *bug trackers* em projetos de desenvolvimento de *software*. Tal problema é caracterizado pela submissão de dois ou mais *bug reports* que descrevem o mesmo problema referente a um *software*, tendo como principais conseqüências a sobrecarga de trabalho na busca e análise de *bug reports*, e o mal aproveitamento do tempo destinado a essa atividade.

Palavras-chave: relatos de bug, gerenciadores de relatos de bug, relatos de bug duplicados, requisição de mudança, experimento, estudo de caracterização, ferramenta, busca

Abstract

Software maintenance and evolution are characterised by their huge cost and slow speed of implementation. Yet they are inevitable activities – almost all software that is useful and successful stimulates user-generated requests for change and improvements. Sommerville is even more emphatic and says that software changes is a fact of life for large software systems. In addition, a set of studies has stated along the years that software maintenance and evolution is the most expensive phase of software development, taking up to 90% of the total costs.

All those characteristics from software maintenance lead the academia and industry to constantly investigate new solutions to reduce costs in such phase. In this context, Software Configuration Management (SCM) is a set of activities and standards for managing and evolving software; SCM defines how to record and process proposed system changes, how to relate these to system components, among other procedures. For all these tasks it has been proposed different tools, such as version control systems and bug trackers. However, some issues may arise due to these tools usage, such as the dynamic assignment of a developer to a bug report or the bug report duplication problem.

In this sense, this dissertation investigates the problem of bug report duplication emerged by the use of bug trackers on software development projects. The problem of bug report duplication is characterized by the submission of two or more bug reports that describe the same software issue, and the main consequence of this problem is the overhead of rework when managing these bug reports.

Keywords: bug reports, bug trackers, bug report duplication, change request, tool experiment, bug report duplication characterization study, bug report search and analysis tool

Contents

1 Introduction **2**

1.1 Motivation . . . 3
1.2 Problem Statement . . . 5
1.3 Overview of the Proposed Solution . . . 5
1.3.1 Context . . . 5
1.3.2 Outline of the Proposal . . . 8
1.4 Out of Scope . . . 8
1.5 Statement of the Contributions . . . 8
1.6 Organization of the Dissertation . . . 9

2 Software Configuration Management **11**

2.1 Introduction . . . 11
2.2 SCM General Concepts . . . 13
2.3 Change Management Overview . . . 14
2.4 Summary . . . 15

3 The State of the Art **16**

3.1 Introduction . . . 16
3.2 Bug Reports Similarity . . . 17
3.2.1 Automated Support for Classifying Software Failure Reports . . . 17
3.2.2 Assisted Detection of Duplicate Bug Reports . . . 18
3.2.3 Detection of Duplicate Defect Reports Using Natural Language Processing . . . 18
3.2.4 An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information . . . 19
3.3 Dynamic Assignment . . . 20
3.4 Evolution and Traceability . . . 21
3.5 Impact and Effort Analysis . . . 22
3.6 Bug reports Quality . . . 22
3.7 Summary . . . 23

4 The Bug Report Duplication Problem **24**

4.1 Introduction . . . 24
4.2 Definition of the Study . . . 25

4.3 Projects and Data Selection . . . 30
4.4 Study Execution . . . 31
4.5 Analysis and Interpretation . . . 32
4.5.1 Question 6: What are the possible factors that could impact on the bug report duplication problem? . . . 36
4.5.2 Main Findings on The Bug Report Duplication Problem . . . 41
4.6 Lessons Learned . . . 42
4.7 Threats to Validity . . . 43
4.8 Summary . . . 43

5 BAST: Bug Report Analysis and Search Tool **45**
5.1 Introduction . . . 45
5.2 The Set of Requirements . . . 46
5.2.1 Functional Requirements . . . 46
5.2.2 Non-Functional Requirements . . . 47
5.3 Tool Architecture Overview . . . 47
5.4 Architecture Components . . . 48
5.5 Bug Report Analysis and Search Tool (BAST) Search Features . . . 52
5.5.1 Ranking and Indexing – Vector Space Model . . . 52
5.5.2 Queries . . . 53
5.5.3 BAST User Interface . . . 54
5.6 Implementation . . . 54
5.7 BAST in Action . . . 55
5.8 BAST Advantages over Other Tools . . . 56
5.9 Summary . . . 57

6 Case Study at C.E.S.A.R. **58**
6.1 Introduction . . . 58
6.2 Definition . . . 58
6.3 Planning . . . 60
6.4 Result Analysis . . . 61
6.4.1 Analysis of the First Treatment . . . 62
6.4.2 Analysis of the Second Treatment . . . 63
6.4.3 Analysis of the Whole Period . . . 66
6.4.4 Analysis Conclusion . . . 67
6.5 Lessons Learned . . . 68

6.6 Summary 69

7 BAST Empirical Evaluation Experiment **70**
7.1 Introduction 70
7.2 Definition 71
7.3 Planning 72
7.4 Operation 77
7.5 Analysis and Interpretation 78
7.5.1 Quantitative analysis 78
7.5.2 Qualitative analysis 82
7.5.3 Lessons Learned 82
7.5.4 Conclusion 83
7.6 Summary 84

8 Concluding Remarks and Future Work **85**
8.1 Research Contribution 86
8.2 Future Work 87

Bibliography **88**

Appendices **94**

A Experiment Instruments **95**
A.1 Time sheet 95
A.2 Questionnaire for Subjects Profile 96
A.3 Form for Qualitative Analysis 97

B Bug-reports Used in the Experiment **98**
B.1 First List of Bug-reports Used in the Experiment 98

C Correlation Graphics **106**

List of Figures

1.1 Example of a bug report 4
1.2 RiSE Labs Influences 6
1.3 RiSE Labs Projects 7

2.1 General change process workflow 14

4.1 Scenarios for rework in bug-repositories. Rework is represented in dark blocks. 25
4.2 Bug reports grouping. 35
4.3 Absolute duplication ratio = duplicate bug reports/total bug reports. . . 35
4.4 Duplication ratio = duplicate bug reports/total bug reports. 36
4.5 Duplication, staff size, and submitters. The values for submitters and staff size on the right side of the charts were reduced using Log10 for better visualization and understanding. 37
4.6 Duplication and software size (Lines of Code (LOC)). 38
4.7 Duplication and software life-time. 39
4.8 Duplication and bug repository size 40
4.9 Submitter profiles and their contribution to duplication problem. 40

5.1 General Text Mining Architecture 49
5.2 General Text Mining Architecture 50
5.3 BAST Architecture 51
5.4 BAST in Action 55

6.1 Repository Status in First treatment 62
6.2 Duplicates found in the baseline tool and BAST in the first treatment . . 63
6.3 Time spent in searches for duplicates in first treatment 63
6.4 Repository Status in the Second treatment 64
6.5 Duplicates found in the baseline tool and BAST in Second treatment . . 65
6.6 Time spent in searches for duplicates in second treatment 65
6.7 Repository status 66
6.8 Duplicates found in the baseline tool and BAST 67
6.9 Time spent in searches 67

7.1 Experiment design 75

7.2 Box plot for time spent on analysis 80
7.3 Box plot for duplicates avoided 80

C.1 Correlation among experience and dependent variables 107
C.2 Correlation among projects and dependent variables 108
C.3 Correlation among bug-trackers and dependent variables 109

List of Tables

2.1 Conducted studies along the years about software maintenance costs (Koskinen, 2004). 12

4.1 Projects characteristics. The life-time is specified in years. 30
4.2 Questionnaire for bug report submitters. 32
4.3 Metrics summary. Maximum and minimum values are in bold and italic respectively. 33

7.1 Subjects profile. 78
7.2 Collected data during the experiment. 79
7.3 Descriptive statistics. 79
7.4 T-tests applied with 95% of confidence to collected data. 81
7.5 Matrix of correlation. 81

A.1 Time sheet used in the study. 95
A.2 Questionnaire for bug-report submitters. 96
A.3 Questionnaire for qualitative analysis. 97

B.1 First list of bug-reports used in the experiment 98
B.2 Second list of bug-reports used in the experiment 102

Acronyms

AJAX	Asynchronous JavaScript and XML
BAST	Bug Report Analysis and Search Tool
BTT	Bug Report Tracker Tool
BRN	Bug Report Network
CCB	Change Control Board
C.E.S.A.R.	Recife Center For Advanced Studies and Systems C.E.S.A.R. (http://www.cesar.org.br) is a CMMi level 3 company with around 700 employees
FR	Functional Requirement
GQM	Goal Question Metric
LOC	Lines of Code
NFR	Non-Functional Requirement
NLP	Natural Language Processing
ORM	Object-Relational Mapper
RiSE	Reuse in Software Engineering Group http://www.rise.com.br
SCM	Software Configuration Management
SD	Standard Deviation
TF-IDF	Term Frequency-Inverse Document Frequency
UFPE	Federal University of Pernambuco
VSM	Vector Space Model
WAD	Work as Design
XP	eXtreme Programming

1 Introduction

Um passo à frente e você não está mais no mesmo lugar
One step forward and you are not in the same place
—CHICO SCIENCE (Um Passeio No Mundo Livre, Afrociberdelia)

Software maintenance and evolution are characterised by their huge cost and slow speed of implementation. However they are inevitable activities – almost all software that is useful and successful stimulates user-generated requests for change and improvements (Bennett and Rajlich, 2000). Sommerville (Sommerville, 2007) is even more emphatic and says that software changes is a fact of life for large software systems. In addition, a set of studies (Huff, 1990; Moad, 1990; Eastwood, 1993; Erlikh, 2000) has stated along the years that software maintenance and evolution is the most expensive phase of software development, taking up to 90% of the total costs.

All of these characteristics from software maintenance leaded the academia and industry to investigate constantly new solutions to reduce costs in such phase. In this context, Software Configuration Management (SCM) is a set of activities and standards for managing and evolving software, defining how to record and process the proposed system changes, how to relate these to system components, among other procedures. For all these tasks, it has proposed different tools, such as version control systems and bug trackers (Sommerville, 2007). However, some issues may arise due to these tools usage. In this work, the focus are the issues from bug trackers, as it will be discussed along this dissertation.

The remainder of this chapter describes the focus of this dissertation and starts by presenting its motivation in Section 1.1 and a clear definition of the problem in Section 1.2. An overview of the proposed solution is presented in Section 1.3, while

Section 1.4 describes some related aspects that are not directly addressed by this work. Section 1.5 presents the main contributions and, finally, Section 1.6 describes how this dissertation is organized.

1.1 Motivation

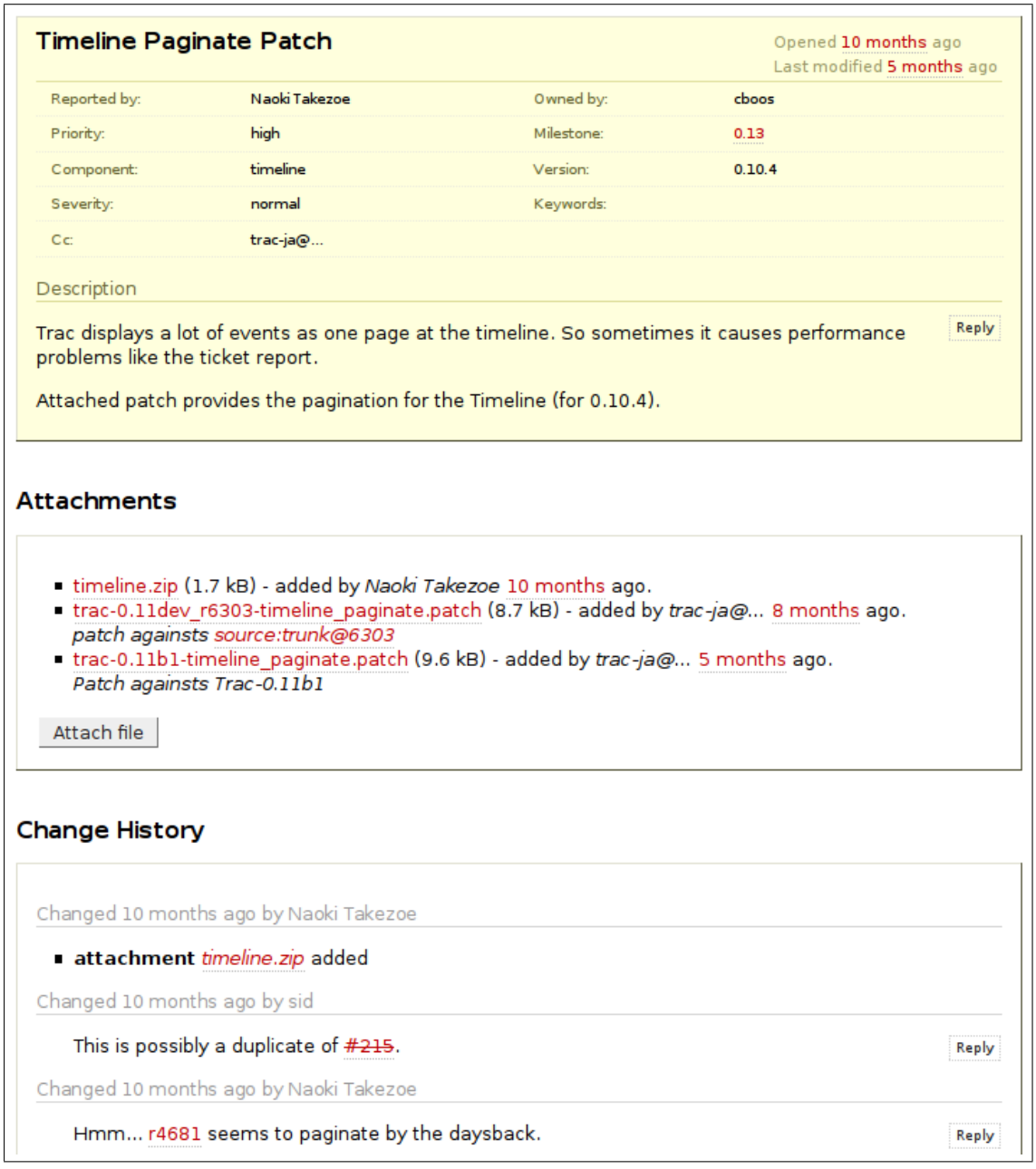
Aiming to improve change management processes, some organizations have used specific systems (generally called *bug-trackers*) to manage, store and handle change requests (also known as *bug reports*). A bug report is defined as a software artifact that describes some defect, enhancement, change request, or an issue in general, that is submitted to a bug tracker; generally, bug report submitters are developers, users, or testers. Such systems are useful because changes to be made in a software can be quickly identified and submitted to the appropriate people (Anvik *et al.*, 2005).

Moreover, the use of bug trackers helps to monitor the software evolution, because bug reports are recorded in a database as well as people involved in a particular bug report are recorded. Thus, changes and their respective responsible can be easily found. Organizations also use such systems to guide the development of software, thus any task to be undertaken in the software development process must be registered and monitored through a bug-tracker. In addition, the historical data of these systems can be used as history and documentation for the software. Examples of such systems are Bugzilla (`http://www.bugzilla.org`), Mantis (`http://www.mantisbt.org`) and Trac (`http://trac.edgewall.org`).

Each bug report is stored with a variety of fields of free text and custom fields defined according to the necessity of each project. In Trac, for example, it is defined fields for summary and detailed description of a bug report. In the same bug report it can also be recorded information about software version, dependencies with other bug reports (duplicate bug reports, for example), the person who will be assigned to the bug report, among other information. Moreover, during the life cycle of a bug report, comments can be inserted to help solving it. Figure 1.1 shows an example of a bug report from Trac.

Some challenges have emerged through the use of bug trackers, among them, we can cite: dynamic assignment of bug reports (Anvik *et al.*, 2006), change impact analysis and effort estimation (Song *et al.*, 2006), quality of bug report descriptions (Ko *et al.*, 2006), software evolution and traceability (Sandusky *et al.*, 2004), and duplicate bug reports detection (Hiew, 2006). Each one of these issues are briefly described as follows:

- **Dynamic assignment** of bug reports is to detect (automatic or semi-automatically) the best developer suited to solve a problem reported in a bug report;



Timeline Paginate Patch

Opened 10 months ago
Last modified 5 months ago

Reported by:	Naoki Takezoe	Owned by:	cboos
Priority:	high	Milestone:	0.13
Component:	timeline	Version:	0.10.4
Severity:	normal	Keywords:	
Cc:	trac-ja@...		

Description

Reply

Trac displays a lot of events as one page at the timeline. So sometimes it causes performance problems like the ticket report.

Attached patch provides the pagination for the Timeline (for 0.10.4).

Attachments

- timeline.zip (1.7 kB) - added by *Naoki Takezoe* 10 months ago.
- trac-0.11dev_r6303-timeline_paginate.patch (8.7 kB) - added by *trac-ja@...* 8 months ago.
  patch against source:trunk@6303
- trac-0.11b1-timeline_paginate.patch (9.6 kB) - added by *trac-ja@...* 5 months ago.
  Patch against Trac-0.11b1

Attach file

Change History

Changed 10 months ago by Naoki Takezoe

- **attachment** *timeline.zip* added

Changed 10 months ago by sid

This is possibly a duplicate of ~~#215~~.

Reply

Changed 10 months ago by Naoki Takezoe

Hmm... r4681 seems to paginate by the daysback.

Reply

Figure 1.1 Example of a bug report

- **Change impact analysis** and **effort estimation** focus on calculating the impact of a bug report in a project and calculating the necessary effort to solve it;
- **Quality of bug report descriptions** is to ensure that the submitted bug reports are properly described;
- **Software evolution traceability** is concerned with the understanding what drives the changes performed in the software along the time; and
- **Duplicate bug reports detection** consists in avoiding the submission of bug reports that describe an already submitted issue.

The focus of this work is trying to avoid duplicate bug reports submission. The problem of bug reports duplication is better explained and characterized in Chapter 4, through a study which examines the factors that cause it and how it impacts on the software development. Furthermore, the other challenges are further detailed on Chapter 3, where it is described the state-of-the-art of mining bug report repositories.

1.2 Problem Statement

The goal of this dissertation can be stated as follows:

> *This work investigates the problem of bug report duplication emerged by bug trackers, characterizing it empirically to understand its causes and consequences, and provides a tool for search and analysis of bug reports to reduce the effort spent on such tasks.*

1.3 Overview of the Proposed Solution

In order to reduce the effects of the bug report duplication problem, it was developed the Bug Report Analysis and Search Tool (BAST). The remainder of this section describes the context where it was developed and the outline of the proposal.

1.3.1 Context

This dissertation is part of the Reuse in Software Engineering Group (RiSE) (Almeida *et al.*, 2004), formerly called RiSE Project, whose goal is to develop a robust framework for software reuse in order to enable the adoption of a reuse program. However, it is

influenced by a series of areas, such as software measurement, architecture, quality, environments and tools, and so on, in order to achieve its goal. The influence areas are depicted in Figure 1.2.

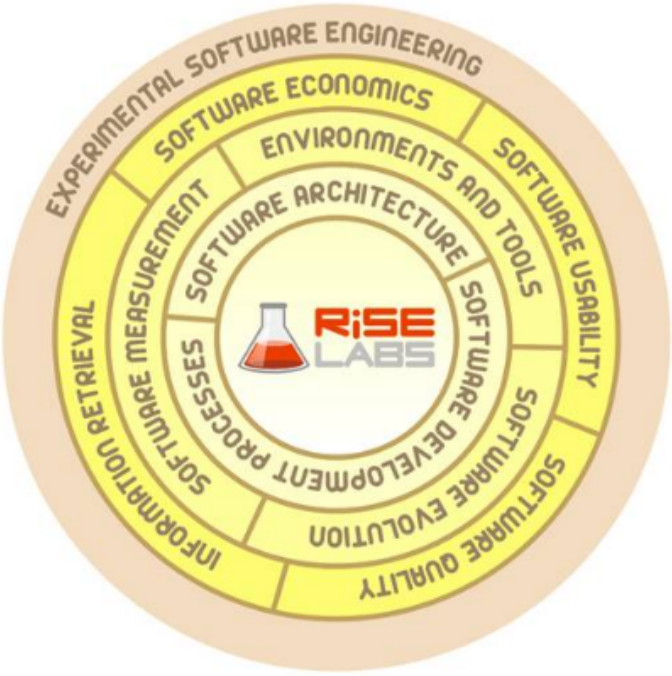


Figure 1.2 RiSE Labs Influences

Based on these areas, the RiSE Labs is divided in several projects, as shown in Figure 1.3. As it can be seen, this framework embraces several different projects related to software reuse and software engineering. They are:

- **RiSE Framework:** Involves reuse processes (Almeida *et al.*, 2004; Nascimento, 2008), component certification (Alvaro *et al.*, 2006) and reuse adoption process (Garcia *et al.*, 2008).
- **RiSE Tools:** Research focused on software reuse tools, such as the Admire Environment (Mascena, 2006), the Basic Asset Retrieval Tool (B.A.R.T) (Santos *et al.*, 2006), which was enhanced with folksonomy mechanisms (Vanderlei *et al.*, 2007), semantic layer (Durao, 2008), facets (Mendes, 2008) and data mining (Martins *et al.*, 2008), and the Legacy InFormation retrieval Tool (LIFT) (Brito, 2007);
- **RiPLE:** Development of a methodology for Software Product Lines (Filho *et al.*, 2008);

- **SOPLE:** Development of a methodology for Software Product Lines based on services;
- **MATRIX:** Investigates the area of measurement in reuse and its impact on quality and productivity;
- **BTT:** Research focused on tools for detection of duplicate bug reports, such as in Cavalcanti *et al.* (2008). Thus, this work is part of the BTT research group;
- **Exploratory Research:** Investigates new research directions in software engineering and its impact on reuse;
- **CX-Ray:** Focused on understanding the Recife Center For Advanced Studies and Systems (C.E.S.A.R.), and its processes and practices in software development.

This dissertation is part of the Bug Report Tracker Tool (BTT) project and its goal is to provide a tool for search and analysis of bug reports with the objective of avoiding duplicate bug reports submission. This work was conducted inside a group for software reuse research, because the bug report duplication problem is more prone to appear when different parts of software are being reused by different projects. A common case where software reuse implies the submission of duplicate bug reports is when the concept of Software Product Lines (Pohl *et al.*, 2005) approach is used to develop software (Runeson *et al.*, 2007). In this context, different software projects share the same basis components, and if some of these components are defective they will affect all software that use these components, thus increasing the possibility of duplicate bug reports submission.

Figure 1.3 RiSE Labs Projects

1.3.2 Outline of the Proposal

The proposed solution consists in a Web based application that enables people involved with bug report search and analysis to perform such tasks more effectively. Although bug report tracking process involves a complete cycle of finding errors, reporting them, validating, fixing the problems and, finally, releasing the changes, the proposed solution aims to assess only the reporting phase. However, the benefits of improving the reporting phase of bug tracking can be reflected to the other phases also, since the time that is saved in the reporting phase can be used to perform the tasks involved in other phases.

1.4 Out of Scope

- **Quality of search results.** The proposed solution uses a well-known model (Vector Space Model (Salton *et al.*, 1975)) to represent documents and perform searches that better meets our necessity, however it is out of the scope of this work to analyze how efficient is the model. Some discussion involving the efficiency of this model can be found in the work of Salton *et al.* (1975);

- **Impact on other phases of bug tracking process.** Our solution concerns with the reporting phase from bug tracking process. Thus, we are interested on how this phase can be improved by the proposed solution. In this way, it is out of scope the analysis and improvement of other phases;

- **Type of users.** Initially, the subjects of this work can be developers, testers or other stakeholders with some technical background in software development, specially using bug trackers. Thus, it is out of scope to provide a tool that supports all types of users.

1.5 Statement of the Contributions

As a result of the work presented in this dissertation, the following contributions can be highlighted:

A characterization of the bug report duplication problem. It was conducted an extensive study about the duplication problem in order to confirm its existence, and potential causes for bug report duplication.

An analysis of the state-of-the-art for mining bug report repositories. It presents an overview of the work found in the literature that have mined specifically bug report repositories, for all diverse purposes.

A solution for bug reports duplication. It specifies and implements a solution based on *Text Mining* and *Keyword search* techniques (Baeza-Yates and Ribeiro-Neto, 1999), with the objective of to reduce the effects of the bug report duplication problem.

Two empirical studies to validate the proposed solution. This dissertation also presents a case study performed in a real environment for software development and test, and an experiment performed with 18 subjects comparing BAST with a baseline tool.

In addition to the contribution mentioned, some papers presenting the findings of this dissertation were produced:

- Cavalcanti, Y. C., Martins, A. C., de Almeida, E. S., and de Lemos Meira, S. R. (2008a). Avoiding Duplicate CR reports in Open Source Software Projects. In *The 9th International Free Software Forum (IFSF'08)*, Porto Alegre, Brazil.

- Cavalcanti, Y. C., de Almeida, E. S., da Cunha, C. E. A., Pinto, E. R., and Meira, S. R. L. (2008b). The Bug Report Duplication Problem: A Characterization Study. Technical report, C.E.S.A.R and Federal University of Pernambuco.

1.6 Organization of the Dissertation

The remainder of this dissertation is organized as follows;

Chapter 2 In this chapter, it is presented a general overview of SCM and change management process. As bug trackers are tools to assess SCM activities, it is important to contextualize it;

Chapter 3 In this chapter, it is presented the state-of-the-art on mining bug report repositories. It details the work that have addressed the problem of bug report duplication, and briefly describes some research concerning other issues, previously mentioned, about bug trackers;

Chapter 4 This chapter characterizes the problem of bug report duplication. It describes a study conducted with several projects (private and open source projects) in order to understand the potential causes and consequences of the duplication problem;

Chapter 5 This chapter describes the proposed solution (BAST), discussing its details, the functional and non-functional requirements, architecture, implementation, among other aspects;

Chapter 6 It describes a case study conducted in an private organization to evaluate the tool. The tool was compared with a private baseline tool during a real cycle of software tests;

Chapter 7 In this chapter it is described an experiment conducted with 18 subjects to evaluate BAST, comparing it with another baseline tool;

Chapter 8 It concludes the dissertation, summarizing the findings of this work, and discussing possible future work and research areas.

2

Software Configuration Management

Two steps forward and you are not in the same place as before.

—YGUARA (Chico's Thought Evolution)

2.1 Introduction

For Pressman (2004) and Sommerville (2007) changes in software projects are inevitable and consume practically most of the time and cost required to develop a software project. According to a newest study (Erlikh, 2000), the necessary changes in a software project may take up to 90% of total costs. Some of these changes could be: changes in business rules, correction of errors, adapting the software to another environment, or even improvements in the software performance. To emphasize the need for software maintenance and its costs, Table 2.1 shows a summary of studies conducted along the years about the costs involved in software maintenance.

The changes made in software characterize its stages of evolution and maintenance, generally performed to fix errors. However, Sommerville (2007) argues that changes in software projects are not only to correct errors, but mostly they are to change business rules and add new features. Thus, this phase of development (the software changes) can be seen as a spiral process, with requirements definition, design, implementation and testing, and it continues along the existence of the software.

Because of its importance, the changes performed in some software must be carefully planned and implemented so that chaos does not take place in the project. When changes are made without following a minimal formal process, the software becomes skewed, making the management of software development so difficult. For example, when

Year	Total costs	Definition	Reference
2000	>90%	Software cost devoted to system maintenance & evolution / total software costs	Erlikh (2000)
1993	75%	Software maintenance / information system budget (in Fortune 1000 companies)	Eastwood (1993)
1990	>90%	Software cost devoted to system maintenance & evolution / total software costs	Moad (1990)
1990	60–70%	Software maintenance / total management information systems (MIS) operating budgets	Huff (1990)
1988	60–70%	Software maintenance / total management information systems (MIS) operating budgets	Port (1988)
1984	65–75%	Effort spent on software maintenance / total available software engineering effort	McKee (1984)
1981	>50%	Staff time spent on maintenance / total time (in 487 organizations)	Lientz and Swanson (1981)
1979	67%	Maintenance costs / total software costs	Zelkowitz *et al.* (1979)

Table 2.1 Conducted studies along the years about software maintenance costs (Koskinen, 2004).

the changes performed in a software are not registered somewhere, it is difficult to determine which version of the software implements a specific functionality or solve any particular issue. The lack of control over changes also complicates the planning of the project, because the development team does not have a well defined set of activities to be addressed.

In the context of distributed software development, the lack of change management is even more serious. In this case, the development teams and/or individuals are geographically distributed, making more difficult the interaction between developers. Open source projects are examples of distributed development, which makes it clear the need for a systematic control of changes. For example, in an open source project, anyone can suggest a change, report an error, or even modify the software in order to carry out the requests mentioned. Without a systematic control of changes, people responsible for the management of the project will not control the evolution of it.

Based on the context of this work and the importance of change management, this chapter brings general concepts about Software Configuration Management (SCM). The remain of this chapter is organized as follows: Section 2.2 presents general concepts about SCM; in Section 2.3 it is presented concepts about the change management process and systems to aid such activity; and Section 2.4 summarizes the chapter.

2.2 SCM General Concepts

As summarized by Pressman (2004), SCM is a set of activities designed to monitor changes, identifying the products of work that can be modified by establishing relations among them, defining mechanisms to manage the various versions of these work products, controlling the changes imposed and making audit, and preparing reports on the changes made.

Among the work products that can be modified, also called *items of configuration*, we can mention: source code, requirements specification, architecture specification, and even the document of SCM, among others. In general, all information produced throughout the software development process can be an item of configuration. SCM also defines the concept of *baseline*. According to the IEEE (IEEE Std. No.610.12-1990), *a baseline is a specification or product that was formally reviewed and approved to serve as a basis for future development and that can be changed only through formal procedures of change management*.

Five activities can be clearly identified in a SCM process (Pressman, 2004):

- **Identification.** During the activity of identification, it is mapped out which products of software development process should be considered items of configuration. In this activity, each item receives a name, the items can be organized according to object oriented approach, and the relationships among the items should also be identified.

- **Version control.** According to Sommerville (2007), a system version is an instance of a system that differs, in some way, from other instances. Versions of the system may have different functionality, enhanced performance or errors fixing. Version control is just the activity of managing different versions of a software using procedures and tools.

- **Change management.** The activity of managing change is responsible for monitoring requests for new features, correction of errors, improvements in performance, among others. In the process of SCM, the changes are not restricted to source code, but to all items of configuration of a software. This will be discussed in Section 2.3.

- **Auditing.** In the audit activity the items of configuration are reviewed to ensure that they meet the specifications of the development process.

- **Reporting preparation.** In the activity of preparing reports, documents are generated with the state of software configuration, describing: what happened, who made, and what happened if other items of configurations had to be modified.

2.3 Change Management Overview

Change management, which is a sub-activity of SCM, is the process by which the software evolution is managed. As mentioned previously, without a formal process to drive software changes, monitoring the evolution of software is seriously compromised. Generally, a process of change encompasses the activities of analysis of incoming changes, planning the release, implementation of the changes, and release of the software to customers.

Figure 2.1 shows a general process for change management. According to such figure, there is the following workflow: firstly are made change requests (bug fixes, improvements etc.); then it is done the impact analysis (which items of configuration have changed, for example); next, it is time to plan which changes will be held for the next release; the changes are implemented; and finally, the new software version is released to customers.

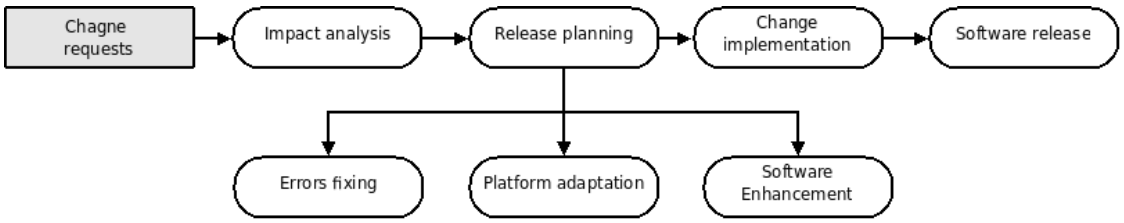


Figure 2.1 General change process workflow (Sommerville, 2007).

The cycle showed in Figure 2.1 is repeated throughout the existence of the software. Moreover, this cycle requires the participation of several people involved directly or indirectly with the software development process. For example, change requests can be requested by users, developers, testers, managers etc. People responsible for the analysis of the changes also need to interact with those who have requested changes to make clear possible doubts. In other words, a process for change management requires synchronization of work and communication among people.

Processes such as RUP (Shuja and Krebs, 2007) and eXtreme Programming (XP) (Succi, 2001) define some activities for change management. However, it is out of scope

of this work to describe the details of such processes. Furthermore, the solution we propose can be adopted in any process.

2.4 Summary

This chapter presented the importance of changes during the software life-cycle and an overview about Software Configuration Management (SCM), which is responsible for leading software evolution. It also described the Change Management activities present in SCM, and presented some aspects and impacts of such activities to the whole development process. Such systems are generally called *bug trackers*, and they act by storing and handling the incoming change requests to some specific software, in order to keep control of the software evolution.

Next chapter presents the state-of-the-art of mining bug report repositories. Thus, it is described some challenges emerged by bug trackers and how researchers have approached them.

3

Mining Bug Report Repositories: The State-of-the-Art

Three steps forward and you are not in the same place as two steps before.

—YGUARA (Chico's Thought Evolution)

3.1 Introduction

There is a variety of work related to mining bug report repositories. However, the work found on the literature are relatively new, dating back from 2003. In general, these types of repositories have been mined for different purposes, such as: *bug reports similarity* (also referenced as duplicate detection), *dynamic assignment of bug reports*, *software evolution and traceability*, *change impact analysis and effort estimation*, and *quality of bug report descriptions*.

All of these proposed categories have the common objective to improve software development, saving costs with software maintenance. Next, we will discuss about each work related to the mentioned purposes. Furthermore, it will be given more attention to technique details involving work related to duplicate bug reports detection (*bug reports similarity*), since this work also addresses such purpose.

Although there are more studies that include mining bug report repositories, they use them to supplement the searches made in other types of repositories such as source code repositories. Thus, we focused our research on work only using repositories of bug reports. For more information about other work not described here, there is a taxonomy

proposed by Kagdi *et al.* (2007b). In such work, it is described several work for mining software repositories, including the ones described here.

The remainder of this chapter is structured as follow: Section 3.2 describes research related to *bug reports similarity*; Section 3.3 presents work involving *dynamic assignment of bug reports*; Section 3.4 describes work concerning *software evolution and traceability*; Section 3.5 discusses *change impact analysis and effort estimation*; and Section 3.6 is about *quality of bug report descriptions*.

3.2 Bug Reports Similarity (duplicate detection)

Duplicate bug reports detection consists on searching for past bug reports to find similar bug reports that describe the same issue as the one being reported, in order to avoid duplicate submission. In that way, the following work (described in chronological order) generally proposes methods to aid such detection.

3.2.1 Automated Support for Classifying Software Failure Reports

Podgurski *et al.* (2003) was the first to investigate bug reports similarity. The bug reports explored in their work were software failures automatically submitted when the software did not work properly. Such reports were composed of information (profile) about state of the software at the time the failure occurred, and possibly with the execution stack trace. This type of reports can raise a common problem encountered by developers: they receive more reports than the time they have available to investigate them. Thus, Podgurski *et al.* proposed an automated support for classifying these reports in order to prioritize and diagnostic their causes.

The proposed approach used supervised and unsupervised pattern classification and multivariate visualization to group together bug reports with closely related causes. The validation of the approach was performed using three projects (GCC, JavaC and Jikes) and the failures were gathered by automated tests. The authors claimed, according to the experiment results, that the approach was effective and scalable.

The main problems with their work is that it was not tested with projects from different contexts (the projects were only compilers), and the types of bug reports are not compatible with the bug reports this dissertation deals with. As mentioned before, such reports are about software execution information, while the bug report that our work treat are related to software error descriptions in natural language.

3.2.2 Assisted Detection of Duplicate Bug Reports

The work performed by Hiew (2006) is closer to ours than the first one described. It investigated the duplication problem caused by natural language bug reports submission.

Hiew proposed to group similar bug reports into *centroids*, thus it would be possible to compare incoming bug reports to the *centroid* with high similarity. In this way, each bug report was processed to compute the value for the Term Frequency-Inverse Document Frequency (TF-IDF) (Baeza-Yates and Ribeiro-Neto, 1999) and placed in the *centroid* with higher similarity. The TF-IDF for a single *centroid* was the combination of the TF-IDF of all bug reports inside the same *centroid*.

Thus, in order to compare the incoming bug reports to each *centroid* it was used the cosine similarity measure (Baeza-Yates and Ribeiro-Neto, 1999). The classification of incoming bug reports were performed labeling them as *unique* or *duplicate* according to the similarity with each *centroid*. A threshold value was specified to decide when an incoming bug report was *unique* or *duplicate*. Moreover, despite to labeling the incoming bug reports, the approach returned a list with higher similar bug reports to enable the submitter to decide when a bug report were correctly classified.

The approach was tested with *21,915* bug reports from Firefox, *37,716* from Eclipse Platform, *1,782* from Apache 2.0 and *22,076* from Fedora Core. These reports were collected between September/2005 and October/2005. In addition, only bug reports classified as *fixed*, *duplicate* and *open* were used in the tests.

The work reported the results for different thresholds, including thresholds for classification and for the length of recommendation list. The approach achieved 29% of precision and 50% of recall at its best. The main difference from this work and our approach is that the technique used in the tool we proposed does not group the most similar bug reports into *centroids*. In addition, our tool is not a recommendation system; the users have to perform search in order to find similar bug reports.

3.2.3 Detection of Duplicate Defect Reports Using Natural Language Processing

Runeson *et al.* (2007) addressed the problem of detecting duplicated bug reports using Natural Language Processing (NLP) techniques. One advantage of such work was the identification of two types of bug reports: 1) those that describe the same problem and 2) those that describe two different problems with the same cause. The former describes the same failure, generally using similar vocabulary, and the latter describes different failures

and may use a different vocabulary. However, Runeson *et al.* restricted their approach to addresses only the type 1.

A tool based on ReqSimile[1] was developed and some operations related to NLP was implemented, such as *tokenization*, *stemming*, *stop-words removal*, *synonyms* and *spell-checking* (Feldman and Sanger, 2007). Moreover, it was used the vector-space model (Baeza-Yates and Ribeiro-Neto, 1999) along with the cosine measure for similarity measurement. Other models were also tested (Jaccard, Dice), but they did not improve the general results.

Tests were performed using the Sony Ericsson Mobile Communications projects, which uses a Software Product Line (Pohl *et al.*, 2005) approach and about 10% of bug reports were duplicated. The methodology to evaluate the NLP technique was composed of two approaches: batch scripts were conducted with selected duplicated bug reports against the database; and experiments with users were conducted followed by interviews.

The result of the work showed that 40% of duplicated bug reports could be found in Sony Ericsson projects. According to Sony Ericsson reports, about 30 minutes are spent to analyze a bug report, what means that for 1000 duplicate bug reports avoided, about 20 hours of analysis time would be saved.

The work showed good results for the *recall* metric, however, the proposed approach cannot be completely analyzed since it does not mention the values for the *precision* metric. Furthermore, it does not discusses the number of bug reports analyzed in the tests, and it does not show details about the experiment with users. Additionally, it does not discusses if there was time saving during the tool adoption.

3.2.4 An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information

In Wang *et al.* (2008), it was proposed an approach to mitigate bug reports duplication problem using NLP and execution information. The execution information is concerned to data about the software execution when the error occurred, such as method calls or variables state. This type of data was combined with natural language data to improve the recall.

The solution proposed is also based on recommending a list of possible duplicated bug reports. The recommendation list was ranked according to a pair of similarities scores: one between natural language from the incoming bug report and existing bug reports;

[1] http://reqsimile.sourceforge.net

and other between the execution information from incoming bug report and existing bug reports.

In order to validate the approach it was performed an experiment with Firefox and Eclipse data, resulting in a *recall* of 67%-93% at its best. These results are very acceptable if compared with other related work. However, some points must be outlined to the experiment and the approach itself.

Concerned to the approach, first we must mention that not every bug report has information about execution. Second, the approach is extremely dependent on the programming language in which the software is developed, since the similarity of the execution information is made based on method calls. Such conditions turn the approach not portable to other environments.

Regarding the experiment, the first negative point is about the amount of projects studied. It was used bug reports only from two open source projects, thus not enabling results generalization to other environments. The second negative point is the few amount of bug reports used; related work have used a range of *4,000* to *12,000* bug reports, which is almost 10 to 20 times high. Moreover, it was filtered out invalid bug reports from the data set. We consider this filter not desirable, because invalid bug reports can also be duplicate.

Moreover, since the execution information was manually extracted by reproducing the errors reported in the analyzed bug reports, it is difficult to replicate the experiment with more projects or with more bug reports. It would also be good if they analyzed the *precision* metric, not only the *recall* metric.

3.3 Bug Report Dynamic Assignment

Also known as *Bug report Triage*, this step of the bug report tracking process consists of identifying which is the best developer to solve a new bug report. Several work have used machine learn techniques combined with versioning system data and/or bug report repositories.

Anvik *et al.* (2006) presented an approach for semi-automated bug report assignment. The approach used a machine learning algorithm to a bug report repository to learn the types of bug reports that each developer resolves. The work of Canfora and Cerulo (2006) also proposed a method to bug report assignment, however in such work it was used information from versioning systems combined with bug reports information.

In other work (Anvik and Murphy, 2007), it was compared which type of repository

(versioning systems or bug report repositories) is better to assign the best developer to a bug report. The work concluded that using bug report repositories is better if the objective is to determine the expertise group with less false positives (developers that are not expert in the given subject), while versioning systems are better for retrieving all experts for a given problem (in this case false positives can occur).

3.4 Software Evolution and Traceability

Mining bug repositories for software evolution and traceability is concerned with understanding what drives the changes performed in the software along the time. Software traceability often involves documents, source code, bug reports, among other assets. Generally, the research related with this purpose combine data from source code repositories and bug report repositories.

Sandusky *et al.* (2004) conducted an empirical research about Bug Report Networks (BRNs) in open source projects. According to them, a BRN is created when members of a software development project assert duplication, dependency, or reference relationships among bug reports. They pointed that BRNs understanding can be useful for decreasing cognitive and organizational effort, refined representations of software and work-organization issues, and rearrange the relationships among project members.

Antoniol *et al.* (2005) proposed a framework to merge information from bug report repositories, source code, and versioning systems. Such framework aids the developer to browser and navigate through the information provided by such artifacts in an interconnected way. For example, some developer could use the framework to visualize what bug reports were fixed in a given software version. Furthermore, he/she could visualize what files of source code were modified.

Koponen and Lintula (2006) proposed an approach to integrate versioning systems and bug report repositories. It used data from Apache HTTP Server and Firefox. Koponen and Lintula investigated if the changes in such projects were driven by bug reports. They concluded that only a small percentage of changes were made because of bug reports in Apache HTTP Server. However, in Firefox 60% of changes are guided by bug reports. Moreover, they discovered that developers who performed few changes are more suitable to be guided by bug reports.

There are also other work in the same direction of Koponen and Lintula (2006), such as Kagdi *et al.* (2007a) and Fischer *et al.* (2003a,b). In the first, commits of versioning system were analyzed to verify frequent co-changes sets of artifacts (e.g. source code

and documentation). The second one aimed to understand the software evolution by integrating versioning systems and bug report repositories, thus looking at the bug reports it is possible to determine which commits were performed to solve a specific issue.

3.5 Impact Analysis and Effort Estimation

The impact analysis and effort estimation purposes are related to determine the amount of time, costs and complexity that a bug report needs to be resolved. By achieving such purpose, it is possible to better manage software projects, such as planning releases and costs (Song *et al.*, 2006).

For that purpose of effort estimation, three works were found in the literature: Song *et al.* (2006), Panjer (2007), and Weiss *et al.* (2007). All of them used data from bug report repositories as input for their approach. For impact analysis there is the work of Canfora and Cerulo (2005), where it was explored bug report repositories and versioning systems using information retrieval techniques to predict the impacted source files for a new bug report submission.

3.6 Bug reports Quality

The quality analysis of bug reports is concerned with how submitters are describing software issues on bug reports free-text fields. Such analysis does not impact directly in software development improvement, however, it addresses issues to better describe bug reports and efficiently conduct the bug report tracking.

In Ko *et al.* (2006), it was performed a study with *200,000* bug reports to understand how submitters describe software problems. They discovered that bug reports summary generally describe software entity or behavior and its execution context. Moreover, "*95% of noun phrases referred to visible software entities, physical devices, or user actions*".

In the work of Bettenburg *et al.* (2007), it was performed a survey with Eclipse developers to understand what type of information they use in bug reports and problems found. The "results showed that the steps to reproduce and stack traces are most sought after by developers, while inaccurate steps to reproduce and incomplete information pose the largest hurdles".

3.7 Summary

This chapter presented the state-of-the-art concerning the mining of bug repositories. It was divided in the following categories: *bug reports similarity* (also referenced as duplicate detection) (Section 3.2), *dynamic assignment of bug reports* (Section 3.3), *software evolution and traceability* (Section 3.4), *change impact analysis and effort estimation* (Section 3.5), and *quality of bug report descriptions* (Section 3.6). For each of these work, it was described their goals and methods, the data set used to validate the methods, results, and a critical analysis.

Next chapter presents a characterization study about the bug report duplication problem. It is analyzed some potential causes of the problem and its consequences to software development.

4

The Bug Report Duplication Problem: A Characterization Study

Four steps forward and you are not in the same place as three steps before.

—YGUARA (Chico's Thought Evolution)

4.1 Introduction

Based on the work discussed in the previous chapter, we identified that duplicate bug reports can generate rework in two different scenarios, as illustrated in Figure 4.1. In the first scenario, the submitters should search for similar bug reports in the repository to avoid the submission of duplicates. However, there is also a common situation where bug reports are submitted without making the necessary searches before (dotted line). In the second scenario, it is considered that the bug reports have been submitted, with or without search or analysis of previous bug reports, and a group of people, called Change Control Board (CCB), are allocated into a group to make the analysis of existing bug reports. In order to do this analysis, the CCB also needs to search in the repository to confirm the uniqueness of a bug report or to assign a responsible for it, for example.

In general, the search and analysis of bug reports performed at the first scenario are not sufficient to find similar bug reports, and thus avoid the submission of duplicate entries. Thus, the presence of CCB becomes essential to reduce the amount of bug reports that the developers need to consider. However, even with the CCB participation, some duplicates still end up passing and stored in the repository.

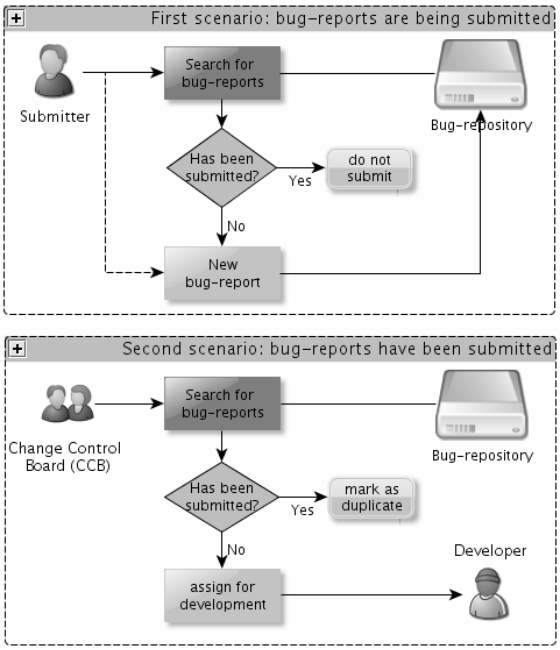


Figure 4.1 Scenarios for rework in bug-repositories. Rework is represented in dark blocks.

This chapter presents a characterization study about bug repositories and search and analysis of bug reports, in order to understand the possible factors that could cause bug report duplication and its impact on software development. Both private and open source projects are used in the study, and different characteristics are analyzed. Furthermore, as a result of the analysis, we can point out what factors can contribute to increase or decrease the bug report duplication problem.

The remaining of this chapter is structured as follows: Section 4.2 presents the definition of the study; Section 4.3 describes the projects and data used in the study. In Section 4.4 is discussed the execution of the study, and in Section 4.5 it is made the analysis and interpretation of the results. Sections 4.6, 4.7 and 4.8 present lessons learned, threats to validity and summary, respectively.

4.2 Definition of the Study

The Goal Question Metric (GQM) method (Basili *et al.*, 1986) was used to define this characterization study. The GQM consists of the definition of the study's goal, the questions to be answered, and the related metrics that must be collected to help answering the questions. For some metrics, we established a baseline based on related work, while others – without previous data available – were explored in this work in order to serve as

basis for future studies.

The goal of this study was to analyze *bug repositories* and the *activities for searching and analyzing bug reports* with the purpose of *understanding them* with respect to the *possible factors that could impact on the duplication problem and their consequences on software development*, from the point of view of the *researcher*, in the context of *software development projects*. The following questions and metrics were defined.

Question 1. *Do the analyzed projects have a considerable amount of duplicate bug reports?* This is the starting point of this study. Although the literature reports that there is a considerable amount of bug duplication in most projects, if the projects analyzed in this study do not have enough duplicate bug reports, the other questions can not be answered properly. Thus, we investigated the projects to find out if they have duplicate bug reports in their repositories, and if the amount of duplicates is large enough to cause problems. The following metric was defined to help answering this question:

- M_1: **percentage of duplicate bug reports in software development projects**. This metric can be collected by analyzing the status of the bug reports in the repository. For example, if the status of a bug report is defined by the keyword DUPLICATE, then it counts as a duplicate. Based on recent work (Anvik *et al.*, 2005; Runeson *et al.*, 2007), we expect an average of 20% of duplicate bug reports present in bug repositories.

Question 2. *Is the submitters productivity being affected by the bug report duplication problem?* As mentioned before, duplicate bug reports have side effects, such as extra time for the analysis. In this context, the productivity is measured in terms of time that is needed to perform bug tracking activities, such as search and analysis of bug reports. We defined three metrics to understand this question:

- M_2: **amount of time spent to search and analyze bug reports before opening a new bug report.** This time is measured in minutes and is counted from the moment a submitter began to investigate (to perform search and analysis of bug reports) a problem until he/she decides whether such problem is new or duplicate. Based on informal interviews with submitters from C.E.S.A.R., we expect values between 10 to 20 minutes for this metric. However, we must note that such values can be biased, since it was not measured empirically;

- M_3: **ratio between the average time to resolve duplicate bug reports and average time to resolve valid bug reports.** For example, if valid bug reports take

x days on average to be resolved, then we wanted to know the percentage (%) of x that is necessary to solve duplicate bug reports. A duplicate bug report is considered solved when it is identified as duplicate and its state on the bug tracker state machine becomes CLOSED. Thus, to calculate M_3, we first divide it into two sub-metrics: (M_3') the average time (in days) to resolve duplicates; and (M_3'') the average time (in days) to resolve valid bug reports. Then we calculate $M_3 = M_3'/M_3''$, which is the ratio. To compute such averages (M_3' and M_3'') we investigated the life-time of each bug report. We did not find any previous data to serve as a baseline for this metric;

- M_4: **average frequency of bug reports per day.** It is the average amount of bug reports that are submitted per day to the projects' repositories. This measure is calculated dividing the amount of bug reports by the total of days that include them. Using this metric, we can analyze how much time is being spent per day with search and analysis of bug reports. We did not find any previous data to serve as a baseline for this metric;

Question 3. *Is there a common vocabulary for bug report descriptions?* The answer to this question is important because it is believed that a controlled vocabulary could help avoiding duplicate bug reports (Lancaster, 1986). For example, with a well defined vocabulary, the submitters could perform better searches using keywords closer from those present in the new bug report.

- M_5: **percentage of common words shared in a bug report group to describe the same problem.** In order to gather confident results, this measure does not count *stop words*[1] of the English language. Moreover, we analyzed only summary and long description fields of the bug reports. Later it is discussed more details about the common words computation. We did not find any previous data to serve as a baseline for this metric;

Question 4. *How are the relationships between master bug reports and duplicate bug reports characterized?* In this work, we consider three types of bug reports[2] in a bug repository: (a) *unique bug report*; (b) *master bug report*; and (c) *duplicate bug report*. A report is classified as unique if it is the only one to describe an issue. If a set

[1] *Stop words* are words that do not improve the searches in Information Retrieval systems.

[2] The definitions for terms (a) and (c) were extracted from the Bugzilla state machine, while the other term was defined by us.

of reports describes the same issue, the first entry will be declared as the master bug report and the others defined as duplicate bug reports. The process where masters and respective duplicate bug reports are bound is called *bug report grouping*. It is important to understand how the groupings are characterized, because it can drive us when choosing adequate techniques to solve the bug report duplication problem. For example, if there are only a few bug report groups, machine learning techniques may not be appropriate to address the duplication problem because the is not sufficient data to train the algorithm.

- M_6: ***grouping* types distribution.** We measured two types of grouping: (M_6') master bug reports that have only one duplicate bug report, also known as *one-to-one* relationships; and (M_6'') master bug reports that have more than one duplicate bug report, also known as *one-to-many* relationships. We could define more levels of relationships, however, we believe that these two are sufficient for our analysis.

Question 5. *Does the type of bug report influence the amount of duplicates?* There are mainly two types of bug reports: *enhancements* and *defects*. Enhancements are normally requests made by users and developers who want new or improved features on some product. Defects are errors or malfunctions reported by users, and normally need to be corrected as soon as possible. Intuitively, it may be argued that defects have more duplicates, because it is more likely that two users will notice the same defect – and report it, and it is less likely that the same enhancement potential is equally perceived by two different users. We wanted to analyze if this is true for the selected projects. If confirmed, this means that defect bug reports require a more careful analysis than enhancement reports, for example.

- M_7: **Duplication ratio = duplicate bug reports / total bug reports.** For each bug report type – enhancement and defect – we calculated the ratio between the duplicate and total bug reports. If this ratio is larger for defects, then this type of bug report causes more duplicates than enhancements.

Question 6. *What are the possible factors that could impact on the bug report duplication problem?* We chose six variables, regarding the projects selected, that we believe could be the factors for duplication or, at least, have some indirect relation to it. These variables are:

- *Staff size.* This variable is related to the number of people involved in the project development. We consider the number of developers as being equal to the number

of people assigned to resolve a bug report. Related work (Anvik *et al.*, 2005) showed that projects with a large staff have many duplicates, thus we wanted to understand if this is also true for projects with a small staff;

- *Number of submitters.* This is related to the amount of submitters in the period that we collected the bug reports. We consider the number of submitters as being equal to the number of people who submitted bug reports in the period. Also in (Anvik *et al.*, 2005), the analyzed projects had many submitters, but we would like to understand if duplicates are also present in projects with few submitters;

- *Software size.* This variable is related to the number of LOC that a software project had until the data selection. The LOC measure, in this study, does not count comments and blank lines. The values for this variable were obtained through the site `http://www.ohloh.net`. We believe that the number of LOC can influence the amount of errors, and consequently more bug reports could be submitted, increasing the chances of duplication;

- *Software life-time.* The software life-time variable is related to the time that a software project has been in development until this study. We computed the life-time of a software from the first bug report submission. Anvik et. al. (Anvik *et al.*, 2005) analyzed projects with 7 and 9 years, however, we must analyze projects with different life-times to understand if it is a factor for bug duplication;

- *Bug Repository size.* This variable is related to the amount of bug reports that a project has in its bug repository. With this variable, we can analyze whether large bug repositories are more susceptible for the submission of duplicates or not. In (Anvik *et al.*, 2005), the bug repositories had a considerable amount of bug reports, however, we wanted to understand if duplicates are present in small bug repositories too;

- *Submitters' profile.* It is important to know what type of submitter profile is more susceptible to submitting duplicate bug reports. The values for this variable are: *sporadic* (**S**), *average* (**A**) and *frequent* (**F**). *Sporadic* is the reporter who submitted at most 10 bug reports in the analyzed period, *average* is the person who submitted between 10 and 30 bug reports in the period, and *frequent* is the person who submitted more than 30 bug reports in the period.

4.3 Projects and Data Selection

To conduct this study, we chose projects from open source organizations and from a private organization. For open source projects, we chose eight (8) projects. For the private project, we chose bug reports from a project being developed at C.E.S.A.R. Regarding the open source projects, we collected all bug reports until the end of June/2008. For the private project, we collected all the bug reports from November/2006 to March/2008, which includes the entire life-cycle for this project. More details about each project are presented in Table 4.1. A brief description of each project is given as follows:

Project	Domain	Code size	Staff size	Bugs	Life-time
Bugzilla	Bug tracker	55K	340	12829	14
Eclipse	IDE	6.5M	352	130095	7
Epiphany	Browser	100K	19	10683	6
Evolution	E-mail client	1M	156	72646	11
Firefox	Browser	80K	514	60233	9
GCC	Compiler	4.2M	285	35797	9
Thunderbird	E-mail client	310K	192	19204	8
Tomcat	Application server	200K	57	8293	8
Private Project	Mobile application	2M	21	7955	2

Table 4.1 Projects characteristics. The life-time is specified in years.

Bugzilla. Bugzilla (`http://www.mozilla.org`) is a software for bug report tracking. This projects is hosted by Mozilla Foundation.

Eclipse. Eclipse (`http://www.eclipse.org`) is an open development platform with support to C, PHP, Java, Python and other languages.

Epiphany. It is the official web browser for, and hosted by, the Gnome desktop (`http://www.gnome.org`).

Evolution. Evolution is a desktop e-mail client. It provides integrated e-mail, address book and calendar features to the users of the Gnome desktop. It is also hosted by Gnome.

Firefox. Firefox is a popular web browser that runs in a variety of platforms (Windows, Linux, Mac etc). Firefox is also hosted by Mozilla.

GCC. GCC (`http://gcc.gnu.org`) is a collection of compilers, including front ends for C, C++, Objective C, Fortran, Ada and Java.

Thunderbird. It is a cross-platform desktop email client hosted and developed by Mozilla Foundation.

Tomcat. Apache Tomcat (`http://www.apache.org`) is an implementation of the Java Servlet and JavaServer Pages technologies, developed by the Apache Foundation.

Private Project. This is a private project being developed at C.E.S.A.R. In this project, applications for mobile devices are developed. It is also a test center for this type of applications.

4.4 Study Execution

This study was performed at C.E.S.A.R., from June/2008 to August/2008. In the beginning of the study, some meetings with projects managers and stakeholders from the private project were conducted. During these meetings, we had discussions about the bug report duplication problem to understand how it was affecting the projects and how to mitigate it. After the meetings, we defined a set of instrumentation assets to be used in the study, such as scripts, time-sheets, and questionnaires. These instruments are described as following:

Scripts. Python scripts were built to read the bug reports from each project. Using these scripts we were able to obtain the values for the metrics M_1, M_3, M_4, M_5, M_6 and M_7. With these scripts we also obtained the 20 most active submitters from the analyzed projects. To identify such submitters, we counted the amount of bug reports submitted by each one of them.

Time-sheets. Time-sheets were used to collect the time to open bug reports by the staff of the private project. A total of four people filled out the time-sheets during a period of two weeks. The time-sheets were applied only in this project because we had access to it and we could monitor this activity. For other projects, this information was gathered through a questionnaire, as described next.

Questionnaire. The questionnaire on Table 4.2 was sent to the 20 most active submitters from each project. We tried to be as simple as possible with the questionnaire.

In total, the questionnaire was sent by e-mail to 180 submitters, with a two-weeks deadline for answer. From these emails, 24 could not be delivered to the recipients, and 141 did not respond the questionnaire in time. Only 17 developers, from 7 projects, replied to the questionnaire: 1 person from Bugzilla; 1 from Evolution, 4 from the Epiphany, 3 from Firefox, 2 from the GCC, 2 from Thunderbird, and 4 from the private project.

The submitters had between 2 and 8 years of experience in the project in which they participated. Moreover, 14 submitters said they spent from 5 to 10 minutes performing searches for bug reports, and only 3 submitters chose the answer 10 to 15 minutes. For the private project, such time was collected using time-sheets and it was between 20 to

Questionnaire
Do you search for past bugs to avoid duplicates? ( ) No, I don't care about duplication. ( ) Yes, I spend about 05 – 10 minutes searching. ( ) Yes, I spend about 10 – 15 minutes searching. ( ) Yes, I spend about 20 – 30 minutes searching. ( ) Yes, I spend more than 30 minutes searching:
Does your project have any technique (automated or manual) to avoid duplicated bug reports? ( ) No. ( ) Yes:
In the case of Bugzilla, do you perform your searches using (Choose more than one if necessary): ( ) Simple web interface – using only keywords ( ) Advanced web interface – with additional parameters ( ) Other:
How do you define your participation on this project? (Choose more than one if necessary) ( ) Developer: ( ) full-time ( ) regular ( ) sporadic ( ) Community manager ( ) Bug Repository Manager ( ) Quality Assurance Manager ( ) User ( ) Contributor ( ) Other:
For how long have you been contributing to this project?

Table 4.2 Questionnaire for bug report submitters.

30 minutes. As mentioned before, such numbers for open source projects can be very biased, since the values were obtained through a questionnaire. It would be more precise if we could apply time-sheets as we did for the private project.

Among the projects, people in the private one spend more time performing searches. This is probably because it is a project dedicated to testing, and testers are advised to try their best to avoid duplicate submissions.

In the questionnaire, it was also asked about techniques to avoid duplicates. Although the projects do not have automatic techniques for detecting duplicates, warnings and guides are placed in bug trackers to alert the submitter about the problem of duplication.

The material used in this study can be accessed through the site `http://www.cin.ufpe.br/~ycc/bug-analysis`. This URL also contains the set of bug reports used in this study, as well as the queries used to extract them from the repositories. We did not publish the private project's data due to confidentially reasons.

4.5 Analysis and Interpretation

This section analyzes the questions defined in Section 4.2, according to the metrics obtained from each project, and presents a descriptive analysis. Table 4.3 summarizes the metrics, presenting the minimum values in *italics*, the ceilings in **bold**, and mean

and Standard Deviation (SD) in the last two columns. Metrics M_6 and M_7 are shown in separate plots, in Figures 4.2 and 4.4, for better visualization.

Metric	*Bugz.*	*Eclip.*	*Epiph.*	*Evol.*	*Firef.*	*GCC*	*Thund.*	*Tomc.*	*Private Proj.*	**Mean**	**SD**
M_1 %	23.32	19.44	31.52	43.24	38.39	17.68	**49.10**	*8.24*	21.59	28.1	13.4
M_2 *(min)*	05-15	–	05-15	05-15	*05-10*	05-15	05-15	–	**20-30**	12.5	1.88
M_3 %	67.69	**75.36**	38.89	54.29	37.04	*32.00*	54.92	48.61	48.25	50.8	14.2
M_4	71	**722**	59	403	334	198	106	*46*	145	231.5	222.1
M_5 %	–	25	–	–	*22*	–	–	–	**35**	31.2	9.5

Table 4.3 Metrics summary. Maximum and minimum values are in bold and italic respectively.

Question 1: Do the analyzed projects have a considerable amount of duplicate bug reports? The values for M_1 in Table 4.3 show a high percentage of duplicates for 8 projects, the exception was the Tomcat project. Only 3 projects were below than the expected (Eclipse, Tomcat and GCC). However, the overall average of duplicates exceeded in 8.1% our expectations. Thus, we can infer that the projects have a high rate of duplicate bug reports.

For projects with major differences below the expected value (GCC and Tomcat), our intuition believe that they receive less duplicates because they have less end users. Generally, people who use GCC or Tomcat are developers with high skills on such projects.

Question 2: Is the submitters productivity being affected by the bug report duplication problem? To answer this question, we analyzed metrics M_2, M_3 and M_4. Every person who answered our questionnaire stated that they are concerned with duplicates and that they perform searches before submitting a bug report. Moreover, according to metric M_2 in Table 4.3, the average time of these searches is 12.5 minutes (note that the mean was computed using the individual responses from reporters). In addition, metric M_4 shows an average of 231.5 bug reports submitted per day.

Thus, if we consider that people who answered the questionnaires are those responsible for most bug report submissions, we have an approximate average of 48 man-hours[3] spent per day only with searches for similar bug reports. So, if we could reduce that time in half by using some techniques to suggest similar bug reports, for example, we would save 24 man-hours per day.

Another important point, shown by the metric M_3, is the fact that a duplicate is resolved using half the time needed to resolve a valid bug report, on average. Thus, reducing the amount of duplicate bug reports coming into a repository would spare more time to resolve valid bug reports. For example, for every two duplicates avoided, a valid

[3]This figure was achieved by multiplying the average of bug reports per day and the average time spent with search and analysis of bug reports: $(231.5\ bugs * 12.5\ min)/60 \approx 48\ man-hours$.

bug report could be resolved. However, it is important to note that we are analyzing time for resolution by the number of days that a bug report takes to be closed. It still would require an analysis of the time spent with each bug report individually to increase the accuracy of that estimate.

Question 3: Is there a common vocabulary for bug report descriptions? We analyzed the descriptions of the bug reports from Eclipse, Firefox and the private project. We were not able to examine the others because their duplicate bug reports did not specify the master bug reports, thus not enabling us to complete the bug report groupings. The values for the 3 projects are considered very close, as shown by metric M_5 in Table 4.3. We do not have previous figures for this metric to determine if this achievement is low or high, however we believe that more words should be shared between duplicate bug reports in order to facilitate the identification.

Since similar bug reports are described with different vocabulary, the identification of duplicates becomes more difficult. So in that context, using a controlled vocabulary may be a way to avoid duplicates. This could be done by defining classes of words that a submitter could use to describe a bug report, as well as to divide the fields of a bug report in more precise information, for example, putting a specific field for execution information *traceback* (Ko *et al.*, 2006; Wang *et al.*, 2008).

Another interesting finding is the fact that the private project's bug reports share more common words than the other ones. We believe it is because the environment in private project is more restricted and controlled, while in distributed open source projects any user can submit a bug report.

Question 4: How are the relationships between master bug reports and duplicate bug reports characterized? Bug report groups present the following characteristics: (M_6') *one-to-one*, a master bug report has a single duplicate; or (M_6'') *one-to-many*, a master bug report has multiple duplicates associated.

Repositories with most groups classified as one-to-many may take advantage of techniques such as clustering or machine learning. This is because such techniques work better when there is more data from where they can learn. For example, if we apply an algorithm of machine learning to recommend possible duplicates when a reporter tries to open a new bug report, firstly this algorithm must run over the existing bug reports in order to extract some pattern from the data and, finally, recommend bug reports that match the pattern of the new one. Thus, if there are many bug report groupings, there are better chances that the algorithm extracts relevant patterns and, then, correctly classifies incoming bug reports.

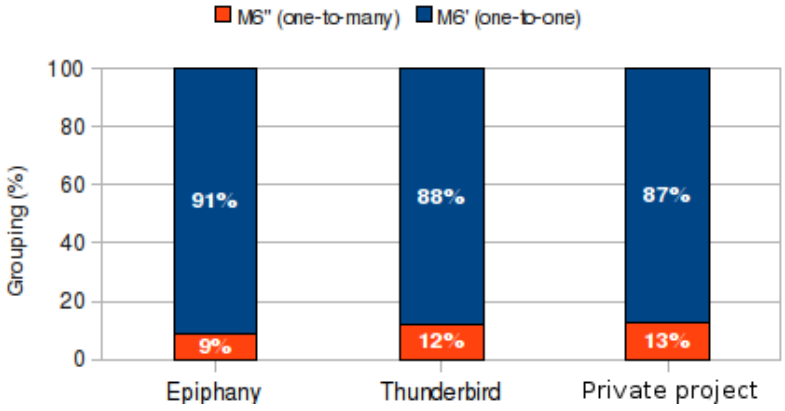


Figure 4.2 Bug reports grouping.

As Figure 4.2 shows, all projects presented in this study have more than 80% of bug report groups characterized as one-to-one. Such situation shows us that it is necessary to investigate more adequate techniques to treat environments where one-to-one relationships predominate. As mentioned before, machine learning and clustering techniques are not adequate for this case.

Question 5: Does the type of bug report influence the amount of duplicates? If we consider only the absolute percentage of duplicate bug reports for each bug report type, we will find out that there are much more defect duplicates than enhancement duplicates, as shows Figure 4.3.

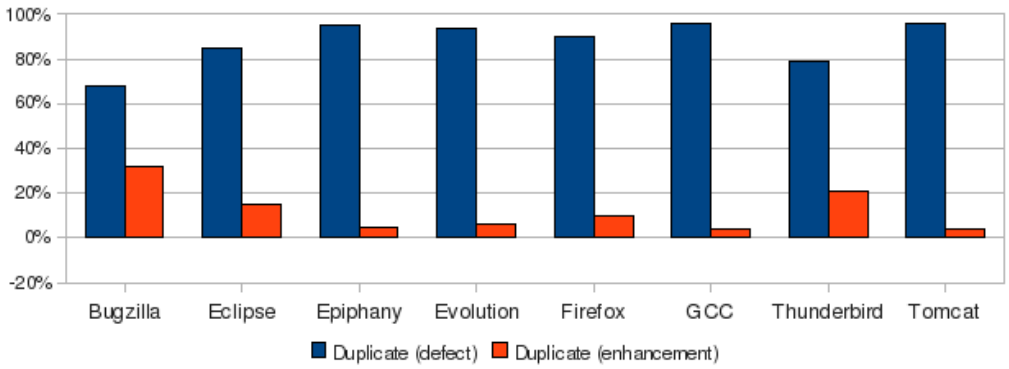


Figure 4.3 Absolute duplication ratio = duplicate bug reports/total bug reports.

But an analysis of the distribution of the bug report types showed that more than 87% of the bug reports are related to defects. Thus, it is natural that there are more defect duplicates in the repository. We needed to understand if this difference also holds when calculating the relative percentage of duplicate bug reports for each bug report type. Figure 4.4 shows the relative duplication ratio for defects and enhancement bug

reports, and their difference in terms of percentages, for each open source project. We did not compute it for the private project because all bug reports from this project are about defects.

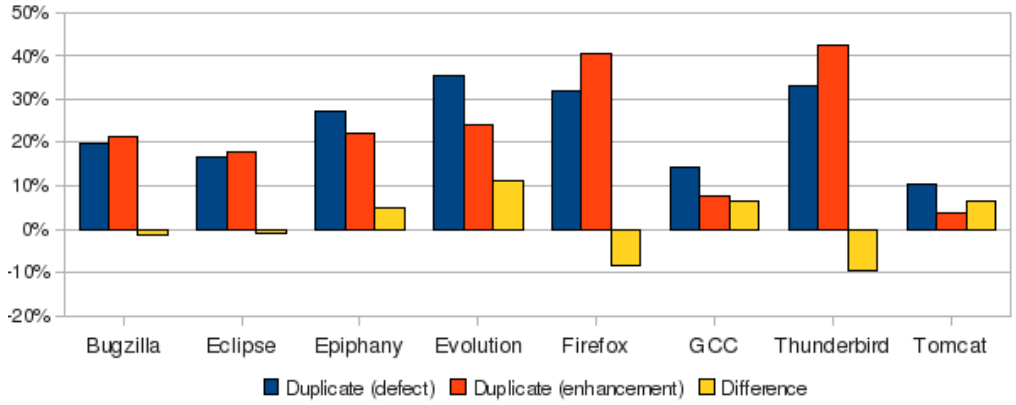


Figure 4.4 Duplication ratio = duplicate bug reports/total bug reports.

As we can see, the ratios are distributed without major differences among the projects. Some projects have relatively more duplicate defects than duplicate enhancements, while in other projects the opposite is true. The mean of the sum of all differences is 3.23%, which indicates that duplicate bug reports for defects are slightly more frequent than duplicate bug reports for enhancements. But this is such a small value, and we can safely conclude that bug reports duplication is not influenced by the bug report type.

4.5.1 Question 6: What are the possible factors that could impact on the bug report duplication problem?

Number of submitters and staff size. Figure 4.5 shows a set of charts showing the relationship between quantity of duplicates, number of submitters, and size of staff. Each chart shows the situation of a specific project. In some graphics, the time does not match the total life-time of the project, because we considered, in that case, the year when the first duplicate bug reports were submitted.

Most projects have a similar tendency in their graph curves regarding the number of submitters and duplicates: the number of duplicates tend to increase together with the number of submitters, but it tends to decrease faster. One explanation for this fact is that people involved in projects get a more uniform vision of the bug repository as time passes, with greater knowledge of bug reports that have already been submitted. Firefox and Epiphany projects exemplify this fall in the number of duplicates without a significant decrease in the amount of submitters.

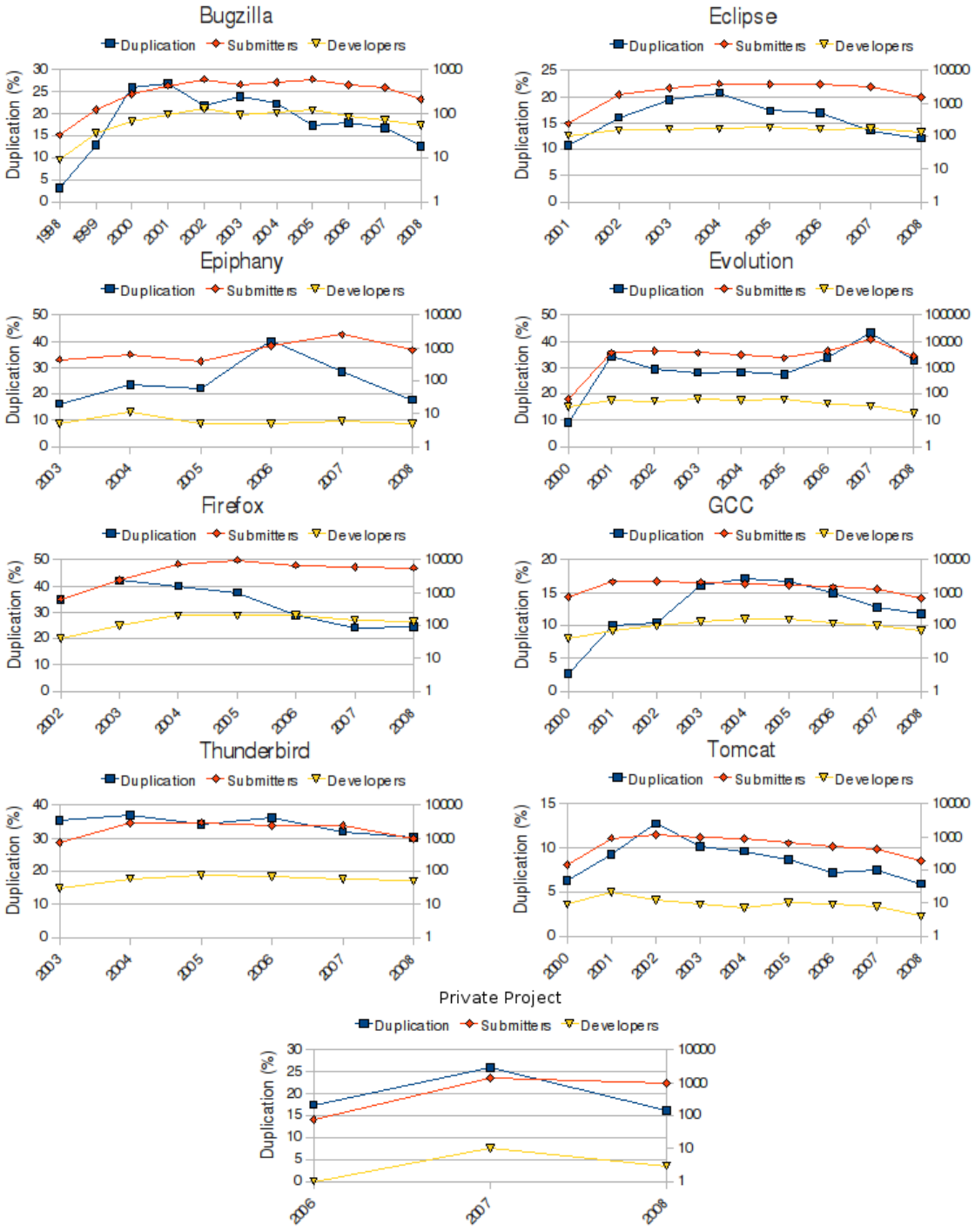


Figure 4.5 Duplication, staff size, and submitters. The values for submitters and staff size on the right side of the charts were reduced using Log10 for better visualization and understanding.

Another interesting fact is that there is always a considerable increase in the number of duplicates in the first year of the projects. This is because the first year of the project often coincides with the first release available to users and testers, from which they start to submit bug reports. For example, the Bugzilla project has 14 years of life-time, but the first duplicate bug report appeared only in 1998, when it launched the first release available to users (version 2.0). This seems also to be true for major changes and new releases, like years 2006 and 2007 for the Evolution project.

Regarding the staff size, it has not suffered considerable changes in almost all projects, and did not show a direct relationship with the amount of duplicates. In contrast, the number of staff was very connected with the number of submitters. After that finding, we calculated that 76% (per project) of the developers are also bug report submitters. From these 76%, an average of 13 submitters are among the 20 most active ones. So, when there is a drop in the number of developers, this drop is also reflected in the amount of submitters.

Software size. According to the graph of Figure 4.6, there is no established pattern that leads us to conclude that there is some relationship between the number of LOC and the number of duplicates of a project. For example, the Eclipse project has the greatest number of LOC but it is the third project with less duplicate bug reports. In another example, the Tomcat project has a number of LOC similar to the Thunderbird project, but the difference in the number of duplicates is about 40%.

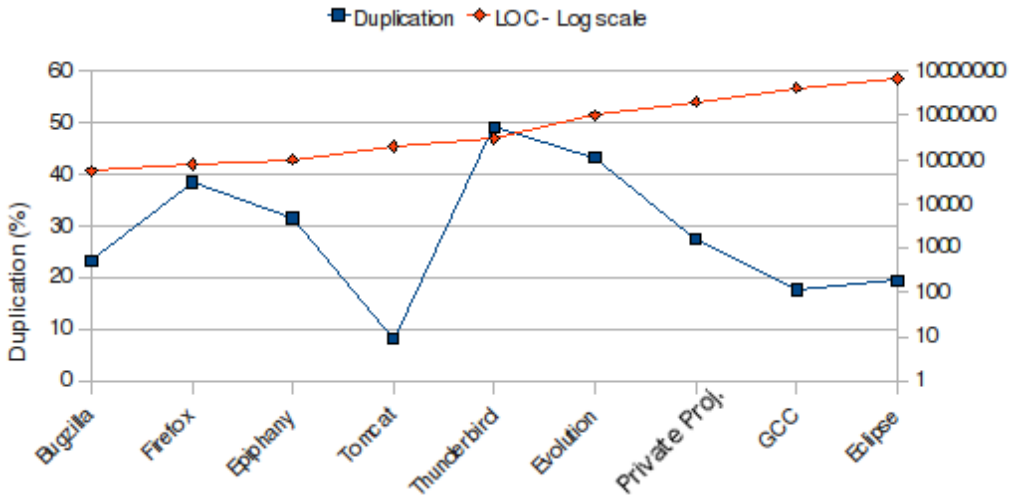


Figure 4.6 Duplication and software size (LOC).

Software life-time. We expected that the life-time was a factor for the duplication, however, as shown in the chart of Figure 4.7, this was not observed in this study. Projects

with higher life-time do not necessarily have more duplicate bug reports than projects with smaller life-time. Thunderbird and Tomcat projects are good examples of this fact; both have similar life-times, but differ sharply on the amount of duplicates. Another example is the Bugzilla and the private project: the first is 12 years older than the second, however it has about 10% less duplicates.

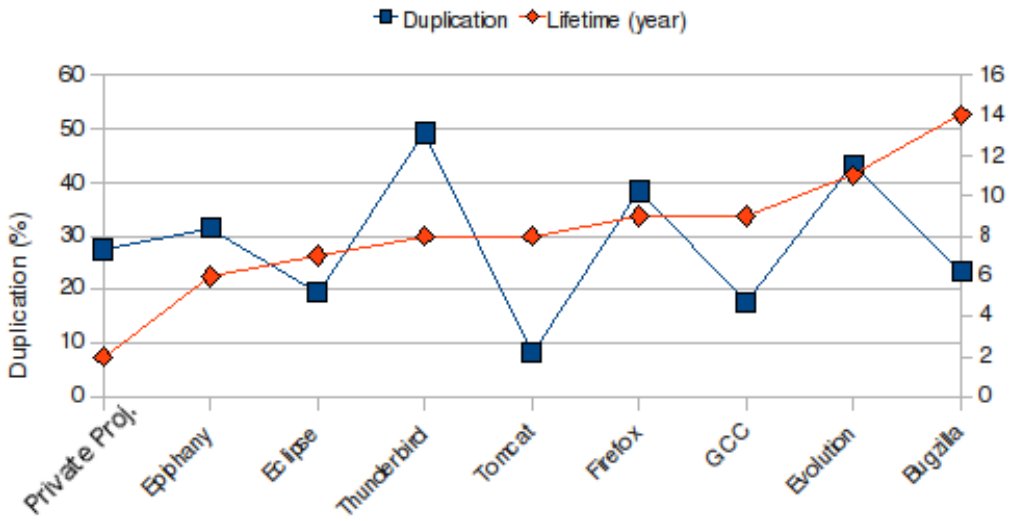


Figure 4.7 Duplication and software life-time.

Bug repository size. The amount of bug reports in repositories is not a factor causing the problem of duplication, as shown in the chart of Figure 4.8. This finding contradicts our initial thoughts. We believed that from the moment that the bug report database increases, it would be more difficult to find similar bug reports. However, we must take into account that as the amount of bug reports increases, the knowledge of the submitters on the repository also increases, which balances the repository growth factor.

Submitter Profile. Figure 4.9 shows the contribution of each type of submitter to the number of duplicates of each project. As it can be seen, most of the duplicate bug reports are submitted by sporadic submitters, followed by average and frequent submitters. The average and frequent submitters contribute very little to the problem of duplication, because these are people who are longest in the project and had good knowledge on the repository.

The sporadic submitters are more prone to submit duplicate bug reports because they have no sufficient knowledge about the project and its bug repository, or they are not aware of the duplication problem. Often these people also have no experience with bug report tracking systems, where the searches must be carried out. Moreover, according to our analysis, most of sporadic submitters submitted at most 1 or 2 bug reports throughout

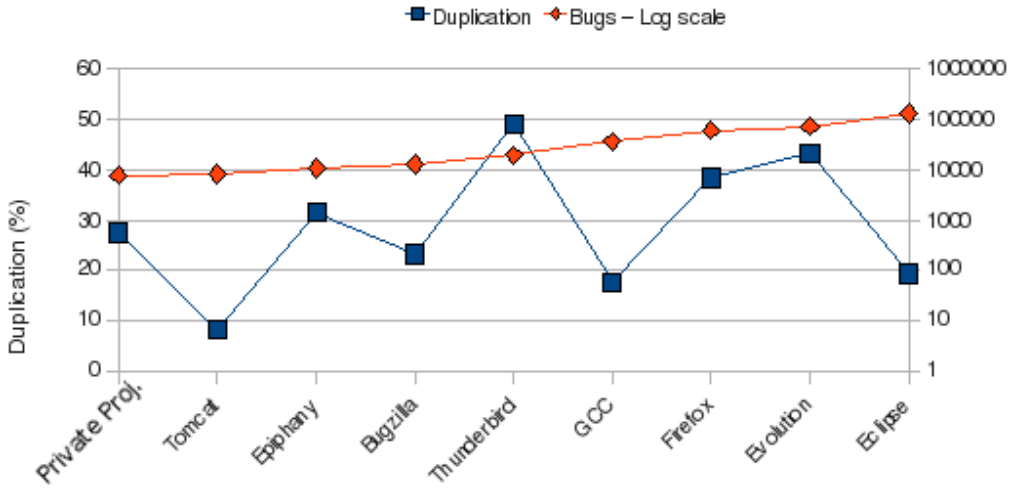


Figure 4.8 Duplication and bug repository size

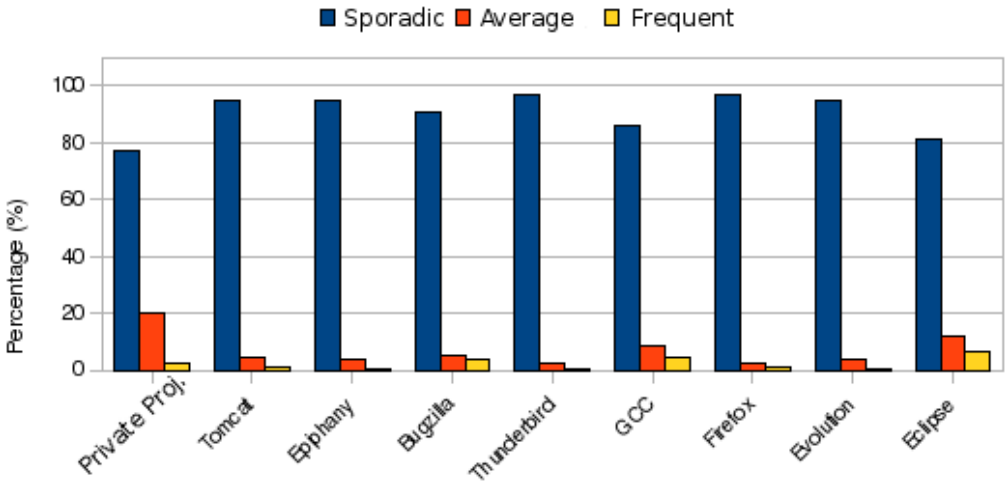


Figure 4.9 Submitter profiles and their contribution to duplication problem.

the life-time of the project.

4.5.2 Main Findings on The Bug Report Duplication Problem

This study was started to understand which characteristics of the projects can be factors for bug report duplication. We also would like to understand if the problem actually exists in the projects examined, and where it impacts on development. The main findings of this study are:

- **All the projects analyzed are being affected by the bug report duplication problem. Some projects are less affected than others, but in general, all of them are affected;**
- **The submitters productivity in the analyzed projects is being affected by the bug reports duplication problem. According to the estimative of this study (taking into account all projects), almost 48 man-hours are necessary each day to do search and analysis of bug reports due to duplication in bug repositories;**
- **The submitters do not use a common vocabulary to describe the content of bug reports. This situation makes it more difficult to identify duplicates. Moreover, none of the projects have explicitly defined a vocabulary to control bug report descriptions;**
- **For the three projects where bug reports grouping were analyzed, more than 80% of the groups are composed by one-to-one grouping type. Thus, clustering and IA techniques may not be applicable to identify duplicates;**
- **The bug report duplication problem can occur independently of the type of bug reports that are being submitted (for example, defect and enhancement bug reports);**
- **The number of LOC is not a factor for the duplication problem;**
- **The size of the repository is not a factor for duplication;**
- **Projects' life-time is not a factor for duplication;**
- **The staff size is not a factor for the duplication problem; and**

- **The profile of the submitter is a determining factor for the submission of duplicates, as well as the amount of submitters. According to our study, inexperienced submitters are more likely to submit duplicate bug reports, as well as a larger quantity of submitters can also increase the amount of duplicates.**

4.6 Lessons Learned

Envisioning a possible replication of this study, we have identified the following aspects that must be improved.

Reporters information gathering. We had difficulties to gather information from open source developers. We selected 180 developers to send the questionnaire, but we received only 13 responses. The reasons for this low number can be many, such as invalid emails or lack of interest. Thus, alternative methods for gathering such information must be developed.

Bug trackers. In our study, we analyzed bug reports from two different bug trackers, but one of them is an internal tool from the private project, without public access. Hence, projects using other types of bug trackers should be included in the analysis. This variation could help us to understand if the bug tracker is a possible factor for the duplication issue.

Private projects. Only one private project was used in this work, which restricted the generalization of the results for that type of project. Thus, future work should include more private projects.

Statistical analysis mechanisms. To analyze the results of this characterization study, we used simple descriptive statistics, such as mean, standard deviation, percentage and dependency analysis. However, we believe that this study is now a starting point for others. New approaches should formalize and test hypotheses based on our results using, for example, probabilistic models.

Negative results. We investigated some aspects for the projects that could be possible causes for the duplication issue, but most of them were not confirmed. We only identified a correlation with the submitters' profile and the amount of submitters. Thus, more in-depth analysis for the negative results must be provided to increase the validity of the results.

4.7 Threats to Validity

The following aspects concerning the validity of the study, and thus its capacity of generalization of the results, were addressed.

Projects population. We analyzed 8 open source projects and 1 private project. The open source projects are quite representative of the population of that type of project, which allows us to generalize the results to the rest of the population with a good degree of confidence.

On the other hand, our analysis of just one private project does not allow us to generalize the results for that type of projects. However, we believe that private projects that have similar characteristics than what we analyzed (i.e.: LOC, submitters, test center, etc) are also being affected by the problem of duplicate bug reports.

Reporters data. The small number of submitters who answered the questionnaire is also a threat to the results of this study. Other ways of gathering information from submitters of open source projects are needed. Although we tried to be straightforward with the questions and used their own email as a tool to obtain the information, it was not enough. Furthermore, the way the time spent on search and analysis of bug reports was collected can be very biased, because it was self reported. In next studies, this data should be collected from more reliable sources, such as time-sheets.

Correlation vs. Causation. We have analyzed several aspects from software projects, bug report submitters and development team, in order to understand correlations between them and the duplication problem. However, we must not confound correlation with causation. The correlations found in the study do not necessary mean a relationship of cause. The causes can be external factors that were not addressed in this work, thus further investigation must be performed.

Open source projects' bug trackers. Except for the private project, all the other analyzed projects use Bugzilla to handle their bug reports. This aspect can impact on the generalization of the results, since the time to search and analyze existing bug reports, as well as the amount of duplicates, may be different in other bug report tracking systems.

4.8 Summary

In this work, we presented a characterization study, using the GQM method (Basili *et al.*, 1986), about the factors that may have impact on the bug report duplication problem, and the consequences of such problem to software projects. The study was performed

using 9 bug report repositories from open source and private projects, with different characteristics, such as software size, staff, life-time, number of bug reports, among others.

According to our analysis, all 9 projects are being affected by the bug report duplication problem. Furthermore, we found evidences that the productivity is being impacted because of the same problem. We also analyzed what project characteristics may be influencing factors to the duplication problem, for example: the number of LOC and life-time have no direct influence on the problem, which contradicts our initial intuition; similarly, increased amounts of bug reports in the repository is not a factor to generate duplicate bug reports.

Next chapter presents the BAST, a tool build to aid the bug report search and analysis. It will be discussed the requirements, architecture, implementation and other aspects.

5

BAST: Bug Report Analysis and Search Tool

Five steps forward and you are not in the same place as four steps before.

—YGUARA (Chico's Thought Evolution)

5.1 Introduction

As seen in the previous chapter, the problem of duplicate bug reports is very critical, mainly because it demands considerable amount of time from submitters and developers. Much of that time is spent with activities of search and analysis of bug reports, or even contacting developers.

In this chapter, we present a tool, called BAST, built in order to facilitate the activities of search and analysis of bug reports, focused on the detection of duplicates bug reports submission. For the development of the tool, functional and non-functional requirements were defined, considering the essence of the herein mentioned activities.

The remainder of this chapter is organized as follows: Section 5.2 presents the requirements of the tool; Section 5.3 shows its general architecture; Section 5.4 describes the components of the architecture; the characteristics of the search engine are presented in Section 5.5; details of the implementation are discussed in Section 5.6; Section 5.7 shows the operation of the tool; and, finally, Section 5.9 presents the summary of the chapter.

5.2 The Set of Requirements

5.2.1 Functional Requirements

According to Sommerville (2007), functional requirements define the functions of a system, which can be, for example, manipulation of data, implementation of algorithms, and the technical details on the implementation of the system. In the BAST specification, the following functional requirements were defined:

- **FR1 - Keyword-based search**. The tool must provide search features based on keyword search, as in web search engines (Baeza-Yates and Ribeiro-Neto, 1999). Moreover, since people are familiar with web search engines, the tool adoption can be easier;
- **FR2 - Rank search results based on bug reports similarity rate.** The search results must be ranked according to the similarity of textual description of bug reports. We believe that textual descriptions bring the most useful information about bug reports. Furthermore, related work (Jalbert and Weimer, 2008) has used such information in their approaches successfully;
- **FR3 - Index bug reports from XML files**. The tool needs to be loaded with bug reports from the bug repositories. For this, one of the options is crawling bug reports in XML files that were exported from such bug repositories. This option requires someone who regularly must feed the system (i.e. a system administrator);
- **FR4 - Index bug reports from original database**. Another option to load the tool is to integrate it directly to the bug repository database. The tool connects to such databases and extract bug reports information from there;
- **FR5 - Extract useful information from bug reports.** After making a search in the tool, the submitters can view each bug report from the result list. Thus, when a bug report is being viewed, the tool must extract relevant information from it, such as related bug reports, external links, execution information, related developers, etc;
- **FR6 - Reports about bug repository status.** Project managers have interest in reports such as: most active submitters; most common duplicate submitters; and bug repository status. All these information help them to better plan project's schedules. Thus, our tool must provide such reports.

5.2.2 Non-Functional Requirements

Non-functional requirements establish conditions that a system must meet to function as desired (Sommerville, 2007). As non-functional requirements, we can cite: accessibility, safety, performance, compatibility with various platforms, and so on. Next are defined the non-functional requirements of the solution:

- **NFR1 - Simple and intuitive filters interface.** One thing that submitters have claimed about is the complexity to create search filters in tools such as Bugzilla, Mantis, Trac and DTTS. Thus, our tool must minimize such complexity in order to facilitate filters creation. Additionally, the study in Chapter 4 showed that most submitters use the advanced search interface of their systems, which bring us the challenge to create a simple search interface the combines advanced features;
- **NFR2 - Integration with most popular bug report tracking systems.** Many organizations have already ran a bug report tracking system with many legacy data, such as Bugzilla and Mantis. Thus, it is useful that our tool can be integrated with such systems to facilitate the tool's adoption;
- **NFR3 - Log search queries and user actions.** In order to perform the evolution of our tool, such as improve the similarity rate, we must collect metrics about the tool's usage. To achieve this, the tool must log all submitters actions;
- **NFR4 - Reasonable similarity rate.** The tool must rank, with reasonable precision, the search results based on bug reports similarity rate;
- **NFR5 - Web-based interface with AJAX**[1]. Submitters are often located in different places, thus the tool must run on a web-server that enables them to access the system independently of their location. Furthermore, the tool's interface must use AJAX to provide better interaction with search results and to decrease the time of the HTTP request/response methods.

5.3 Tool Architecture Overview

BAST's goal is to provide a system for search and analysis of bug reports, thus, the time spent in these activities can be reduced and more duplicates can be avoided. To achieve this goal, the tool will need to meet all the requirements set out previously.

[1] AJAX is an acronym for Asynchronous JavaScript and XML.

BAST is basically composed of three main modules: a module for the Web application (**BAST Web Application**), which submitters interact and do searches and analysis; a module responsible for implementing the features of search and information extraction (**BAST Core**); and a module for storing and managing the data (**BAST Database**).

The way a submitter interacts with the BAST and the architecture of it, follows the way a generic application of *Text Mining*. According to Feldman and Sanger (2007), Text Mining can be defined as an intensive process where a user interacts with a collection of documents for information about a particular subject. Bringing this definition to our context, the users are submitters and the documents are bug reports. The submitters interact with the collection of bug reports in order to examine existing bug reports before submitting new ones. During the analysis, it is very common the analysis of duplicates. To verify whether a bug reports is duplicate, submitters should perform searches in the systems and interact with existing bug reports in order to find similar bug reports.

BAST is an instantiation of a generic application for Text Mining (5.1), however, few modifications were made to accomplish our needs. As can be observed in Figure 5.2, we removed from the original architecture (Figure 5.1) the module *Text Mining Discovery Algorithms*, performed some technical refinement, and reduced the scope of types of documents. To be more specific, we refined this architecture to cover only the parties to search by keyword, indexing, information extraction and visualization of bug reports.

5.4 Architecture Components

The architecture of the tool is composed of a database module, a core module, and a Web interface module. Figure 5.3 illustrates a simplified organization of the architecture of the application: the database stores in an organised way the bug reports from the bug repositories to facilitate further searches; the main module has sub-modules for text processing, indexing of content, parsers, search, and information extractors; and the Web module implements the user interaction features that will be exposed to submitters through a Web browser. Next, it will be provided more details for each component.

BAST Database. It uses the database to store all data from bug reports inserted in the application. We did not use only a simple structure for indexing and retrieval, as provided by Lucene[2], because BAST needs to make complex SQL queries, such as one to draw relationships among bug reports. The technologies used to implement the database were:

- **MySQL.** To implement the database of the application, it was chosen the MySQL.

[2]http://lucene.apache.org

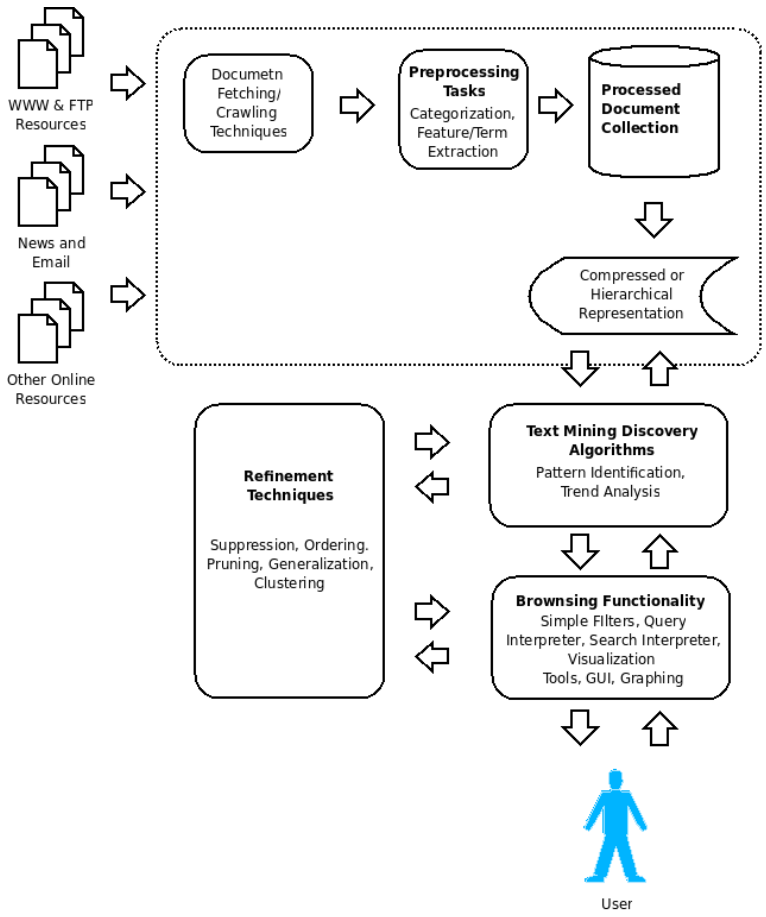


Figure 5.1 General Text Mining architecture (Feldman and Sanger, 2007).

Such database was chosen because it features indexing of content, stability, good performance, good documentation, strong open source community, and it is a free project;

- **SQLAlchemy.** SQLAlchemy is an Object-Relational Mapper (ORM) for Python language[3]. Using an ORM framework, can be possible to change the database used in the tool at any time. For example, we can change from MySQL to PostgreeSQL without breaking the system code. Furthermore, with a ORM tool, it does not need care about SQL syntax because the queries are made using elements of the programming language itself.

BAST Web Application. Running the tool on the Web was one of the main requirements defined for the construction of the tool. Moreover, the Web interface must be constructed in a way to make as simple as possible the use of search and analysis features,

[3] http://www.sqlalchemy.org

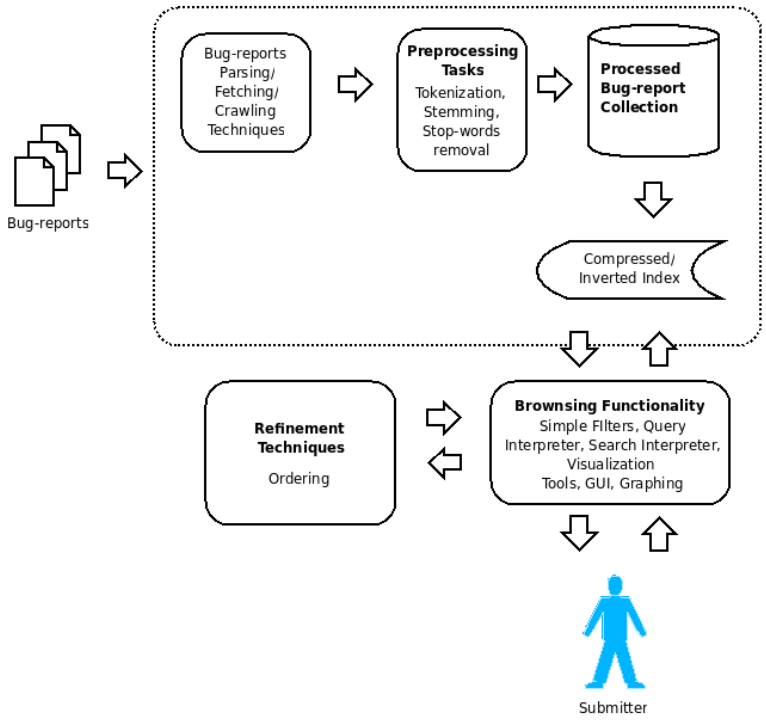


Figure 5.2 General Text Mining architecture instantiation for BAST.

while increasing the efficiency of it. To achieve such purpose, we chose the following tools:

- **HTML/Cheetah.** The interface of the application was made on the Web using HTML and the template tool Cheetah[4]. The Cheetah is used to display dynamic content of the application;
- **DOJO JavaScript Toolkit.** The Dojo[5] framework was used to implement the JavaScript part of the tool. Through DOJO, the creation of user interfaces that make use of AJAX technology becomes easier and more productive;
- **Cherrypy.** Cherrypy[6] is a Web application server for applications written in Python. It was used to implement the responses for the submitter requests on the Web interface.

BAST Core Components. BAST Core module contains the main components of the tool, which are responsible for running the search engine, indexing, information

[4]http://www.cheetahtemplate.org
[5]http://www.dojotoolkit.org
[6]http://www.cherrypy.org

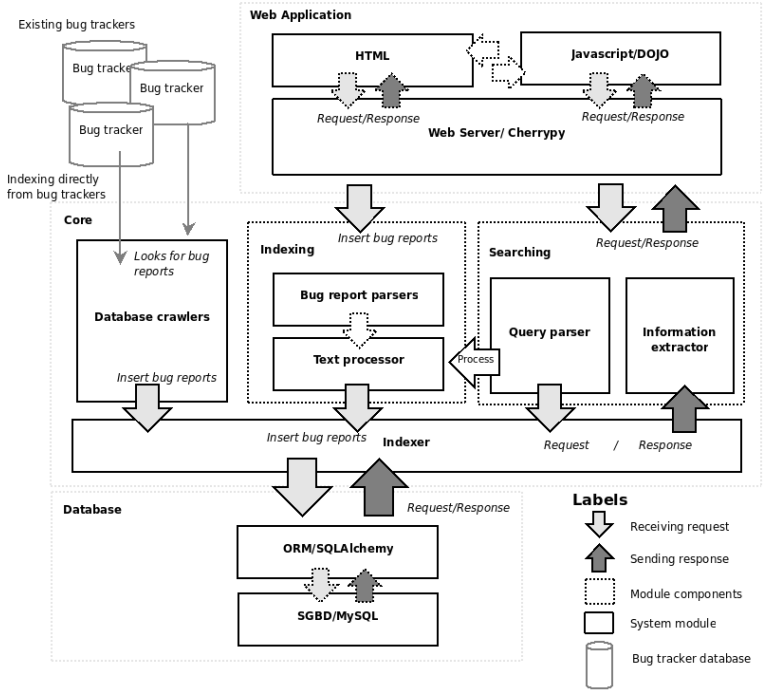


Figure 5.3 BAST Architecture - Web interface, core module and database

extraction and so on. Next, it will be discussed each of these components:

- **Database Crawlers.** The crawlers are used when the tool is connected directly to a database of some existing bug tracker. In this case, the bug reports are directly imported from the bug tracker database. For example, if BAST is connected to an instance of Bugzilla, these crawlers keep watching the database of Bugzilla, and when any change is detected they update the database of BAST of new information;

- **Bug report Parsers.** The parsers are used when the BAST is being loaded by imported XML files from some bug tracker. When these files are sent to the BAST application, it parses the bug reports and includes them in the database of BAST;

- **Text processing.** This module is responsible to process the text from bug reports and search queries. Some of the features of this module are: text tokenization, stop-words removal, and apply Porter algorithm (Porter, 1980) for stemming. The Porter algorithm is used to reduce a word to its radical. For example, the words *tests*, *testing*, and *tested*, are reduced to *test*;

- **Indexer.** The module Indexer is responsible for processing the contents of each bug report in the database before the insertion. It uses the same module for text-

processing to include only relevant information in the index. The Indexer module is also used to update the existing index and perform the searches. Moreover, the index structure is created and maintained by the MySQL engine;

- **Query-parser.** The module Query-parser is used when a submitter inserts a search query in BAST. It processes the query in order to create the filter specified by the submitter;

- **Information Extractor.** This module is used to extract relevant information from a bug report when it is being viewed. For example, when a submitter is analyzing a bug report, it is shown information about related bug reports, external links, and people related to this bug report. This information is extracted using regular expressions and performing queries in the database.

5.5 BAST Search Features

The search for bug reports is one of the main and most important services that BAST provides for the submitters. It is through these searches that the submitters find similar bug reports in order to not submit duplicates. In addition, the tool needs to provide an efficient way to view the results of searches to reduce the time spent on analysis and comparison of bug reports returned. The following subsections describe the mechanisms for indexing and ranking, query, and visualization of searches.

5.5.1 Ranking and Indexing – Vector Space Model

To sort the results of a search, it was used the Vector Space Model (VSM) (Baeza-Yates and Ribeiro-Neto, 1999). With VSM, the documents are represented in a space of vectors of terms. Each dimension (vector) of this space is represented by a term, and this term is associated with a weight known as *Term Frequency-Inverse Document Frequency* (TF-IDF) Salton *et al.* (1975). The symbol TF-IDF is an abbreviation for *Term Frequency-Inverse Document Frequency*, and means that the value of the weight of a term is calculated taking into account the frequency in which this term appears in the collection of documents, and the frequency in which it appears in each document. TF-IDF is calculated by the Formula 5.1, where: $w_{i,j}$ is the term j of document i; $f_{i,j}$ if the frequency of the term j in document i; and n_i is the amount of documents which contain the term j.

$$w_{i,j} = f_{i,j} * \log \frac{N}{n_i} \quad (5.1)$$

Just as the documents are represented in the VSM, then so are the queries. Thus, the VSM proposes to assess the degree of similarity between a document d_i and a query q as a correlation between the vectors $\vec{d_i}$ and $\vec{q}$. According to Baeza-Yates and Ribeiro-Neto (1999), this correlation can be quantified by the cosine of the angle between these two vectors, as shown in Formula 5.2. The VSM ranks the documents according to their degree of similarity with the search query.

$$sim(d_i, q) = \frac{\vec{d_i} * \vec{q}}{|\vec{d_i}| * |\vec{q}|} \quad (5.2)$$

5.5.2 Queries

According to Baeza-Yates and Ribeiro-Neto (1999), a query is the formulation of a user information need. In the case of bug report analysis, submitters type some keywords into the system to search for existing bug reports. We chose to use keyword-based querying because it is intuitive, easy to express, and allow for fast ranking.

In BAST, it is provided *natural language queries* with a combination of a variation of *boolean queries*. Such variation on boolean queries allows submitters to search for complete phrases. For example, if a submitter put some keywords enclosed in quotes (i.e. "*radio fm error*"), the system will return only the bug reports that match the entire query "*radio fm error*", also respecting the order of the keywords.

In the case of natural language queries, all the bug reports matching a portion of the submitter's query are retrieved. Higher ranking is assigned for those bug reports matching more parts of the query. Such type of queries leaves the submitters free to put as much of information as possible about a bug report being searched, raising the chances of finding similar bug reports. An example of natural language query could be: *error when clicking menu button to play music*.

Moreover, BAST implements some search filters that can be inserted directly in queries, improving the use of keywords. For example, the submitters can perform searches by bug reports about specific components or submitted on a specific date. Considering the following query "fm radio error component: driverLG author: joseph@virsys.com", it would be returned only bug reports for errors in the FM radio component *driverLG*, submitted by *joseph@virsys.com*.

5.5.3 BAST User Interface

Another important feature of BAST is its interface with the user. BAST was built in order to facilitate and expedite the tasks of search and analysis of bug reports. So, as well as algorithms for efficient information retrieval, it is also important to provide an interface which the user can interact efficiently and achieve their goals.

The interface of BAST was designed in a way that users can perform their activities on a single screen. For example, often the analysis of bug reports resulting from a search requires a comparison between two or more bug reports. However, current tools, such as Bugzilla and Mantis, do not allow the bug reports to be displayed on the same screen of search results, which means that a submitter needs to switch among different screens (tabs of a browser, for example) to compare bug reports or to select other bug reports from the search result list.

Thus, using BAST, submitters can view the results of the search and content of the bug reports on the same screen, turning more efficient the comparison of bug reports. Moreover, as discussed previously, the tool not only displays the contents of bug reports, as well as extracts and displays relevant information (references to other bug reports, external links, etc) to the submitters. It is important to note that such information are not present in the original database of bug reports.

5.6 Implementation

The system is most implemented using the Python programming language. According to Python Software Foundation (2008), "Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code".

The motivation for choosing Python was its flexibility with text processing, which is an essential task for BAST, and rapid prototype construction. Furthermore, recent work (Prechelt, 2000) has showed that programming in dynamic typed languages is much more productivity than programming in static typed languages, and the number of lines of code in the result system is 2-10 times shorter. Other languages used to developed the tool were JavaScript and SQL.

The BAST tool was developed by only one person and it took about 5 months of

intensive development (30 hours/week). In general, the application contains about 5000 lines of code, with approximately 60% of Python code, 25% of HTML code, 10% of JavaScript code, and 5% of SQL. All components present in BAST are platform-independent, meaning that the application can run on multiple operating systems.

5.7 BAST in Action

In this section, it is shown the operation of the tool and its main screen, how is showed in Figure 5.4. The main fields of BAST are:

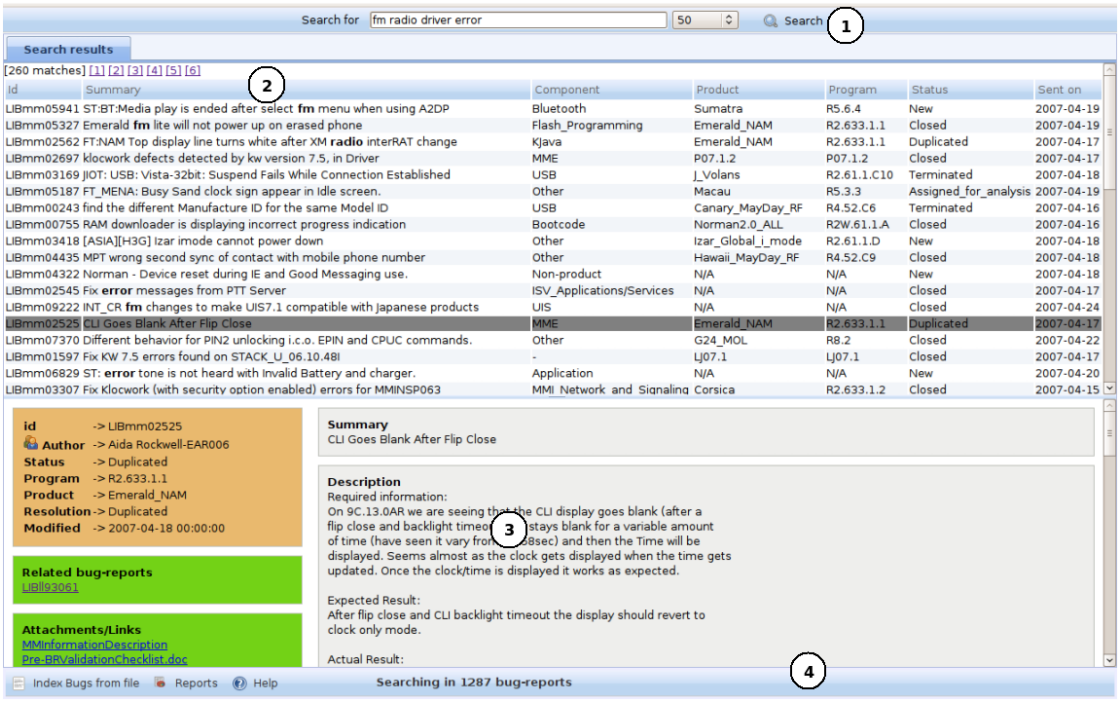


Figure 5.4 BAST in Action: main window.

1. **Search bar.** This is the part of the tool in which the submitters insert queries. It is also in that field where the search filters are specified, as mentioned earlier. In addition to write the query, the submitters can also specify the number of results that must be shown per page. For example, it can be specified that only 30 bug reports should be presented per page;

2. **Search results.** In this field it is shown the results of the searches. Each line represents a bug report, with information about: identification of the bug reports; summary; component; product; program; status; and date of submission. Such

information may change depending on the type of bug report which is indexed in the database. In the case of Figure 5.4, it is shown information about bug reports of the bug tracker from private project of C.E.S.A.R.;

3. **Bug report visualization.** This area of the tool is responsible for showing more detailed information about the bug reports. This information is displayed when a bug report is selected in the list of results. Among the information displayed, it is shown the part of information extraction on left side, where can be viewed: general data, related bug reports, attachments and external links, and related persons. At the right side, there are: the summary, detailed description, and comments. As in the previous item, this information also depends on the type of bug report stored in the database;

4. **Tool bar.** In this part of the tool are presented some information about the environment and some utilities. Through the toolbar, submitters can view information about number of bug reports in the database as well as utilities to import new bug reports and viewing reports. The utility for importing is used when the BAST is not connected directly to a database of a bug tracker. Through this utility, the submitters can upload bug reports into a single compressed file to update the database of the tool. The utility for reports is used to show information about the status of the repository, which submitters submit more duplicates, and the most active submitters, among others.

The numbers in Figure 5.4 also show the common sequence to use the tool. It is, the submitters typically use the BAST according to the following flow: (1) first it is inserted the search keywords and the desired filters, (2) shortly after the search, the submitters examine the results list to identify similar bug reports, (3) and finally, if necessary, the submitters select the bug reports in the results list to view detailed information. Step 4 is optional and can be performed at any time, since there are bug reports indexed in the tool.

5.8 BAST Advantages over Other Tools

BAST implements most of the techniques used by Runeson *et al.* (2007) and Wang *et al.* (2008), such as keyword search and stemming. We did not implement machine learning techniques, as proposed by Anvik *et al.* (2005) and Hiew (2006), because they do not have the performance if compared with cheapest ones, such as the vector model used by Runeson *et al.* (2007). Moreover, we provided evidences on Chapter 4 that the aspects

of bug report data present on repositories do not enable the usage of Anvik and Hiew's proposals.

Beyond the information retrieval techniques implemented, BAST also focuses on usability. The user interface was formulated to improve the user experience, such as to facilitate the analysis of bug reports and searches. We used AJAX techniques to improve the response of the tool, as well as to keep the user focus on bug reports analysis. For example, to compare two similar bug reports, the user does not need to switch among two different tabs from the Internet browser.

5.9 Summary

In this chapter, it was presented a web-based tool for search and analysis of bug reports, with main focus on avoiding duplicates. The tool was built based on the need to facilitate and expedite these tasks in order to reduce the costs of software maintenance. Thus, the application requirements were described as well as the architecture, the components of the architecture, some features for searching and analysing bug reports, details of implementation (programming language and frameworks), and the operation of the application.

Next chapter presents a case study performed inside a private project from C.E.S.A.R. In such study, BAST was used by one subject in order to compare it against a baseline tool in real cycle of tests.

6

Case Study at C.E.S.A.R.

Six steps forward and you are not on the same place as five steps before.

—YGUARA (Chico's Thought Evolution)

6.1 Introduction

In previous chapter, we presented a tool developed with the objective to improve bug-report search and analysis. In this chapter, it will be described a case study conducted inside a private partner organization from C.E.S.A.R., where BAST was tested by one tester during a real cycle for software testing. In this context, BAST was compared with the baseline tool from such organization. Although it was an initial evaluation for BAST, the results were very significant.

Furthermore, this case study served as a pilot project for the experiment that will be described in Chapter 7. The remain of this chapter is organized as follows: Section 6.2 defines this case study; in Section 6.3 the planning of the case study takes place; Section 6.4 shows the analysis and interpretation of the results; Section 6.5 describes the lessons learned during this study; and Section 6.6 summarizes this chapters.

6.2 Definition

Context. During the months of July and August 2008, we performed a case study in a private partner of C.E.S.A.R., to evaluate the tool developed. In this test-center, the testers perform a systematic process to test software that is developed by the same organization in various sites around the world. As the software development is distributed, tests are

also, in many cases, performed in different test-centers. Briefly, the testers receive a formal document with sequences of tests that must be performed, and when errors are found, bug reports should be submitted to report errors.

However, as shown in the study described in Chapter 4, the fact that there are several people testing and developing software distributed, could lead to the submission of duplicate bug reports. In the case of C.E.S.A.R., we know that these bug reports negatively affects the productivity of the team. The productivity of developers is affected, because they will inevitably spend time analyzing bug reports that have already been solved. In the case of testers, productivity is affected because, despite the existence of the duplication problem, they need to concentrate more effort on analyzing similar bug reports in order to prevent the submission of duplicates.

Case study objectives. The macro objective of the study is to analyze the efficiency of the tool in the analysis of bug reports, with focus on duplicates prevention and time saving. Thus, the specific objectives of this case study are: (i) to examine whether our tool can prevent more duplicate bug reports than the current tool of C.E.S.A.R., and (ii) to consider whether our tool decreases the time spent on analysis of bug reports.

Baseline. Currently, the private organization at C.E.S.A.R. uses an internal tool to manage bug reports, as well as to search the existing bug reports. To perform searches in this tool, the submitters need to create multiple filters using a sintax similar to SQL to specify the desired criteria for searches. Some of the negative points were raised by submitters on the current tool: the time and complexity to build the search filters, and the bug reports visualization that did not facilitate the identification of duplicates. Thus, this study compare the results with our tool and the private organization's tool.

Hypotheses. According to the case study, we established the following null hypotheses and alternative ones. According to Wohlin *et al.* (2000), the null hypotheses are those who the experimenter wants to reject, while the alternative hypotheses are those that the experimenter wants to confirm.

Null hypotheses. The null hypotheses determine that the time spent with BAST is greater than the time with baseline tool, and the baseline tool can prevent more duplicates than BAST.

$$H_0: \quad \mu_{time\ with\ BAST} >= \mu_{time\ with\ baseline}$$
$$\mu_{duplicates\ avoided\ with\ BAST} <= \mu_{duplicates\ avoided\ with\ baseline}$$

Alternative hypotheses. For alternative hypotheses, BAST spent less time than the baseline tool and it can also prevent more duplicates than the baseline tool.

$$H_1: \quad \mu_{time\ with\ BAST} < \mu_{time\ with\ baseline}$$
$$\mu_{duplicates\ avoided\ with\ BAST} > \mu_{duplicates\ avoided\ with\ baseline}$$

6.3 Planning

The case study was performed using a real test cycle of specific software for mobile devices. During that period, it was used the internal tool of the private organization and our tool. A specific tester, called *bug report master*, was selected to use both tools in parallel during the whole period. We selected such specif tester because it conducts most of the bug reports analysis and searches, and assists other testers with the same activities.

The *bug report master* is the person responsible for conducting the test cycles. When a tester has problems or questions with regard to testing or a bug report, the *bug report master* should be contacted to solve the problems. Thus, this person, besides being responsible for the submission and analysis of its own bug reports, is also responsible for searching for similar bug reports for people who contacted him. For example, if a tester has doubts related to the uniqueness of a bug report, but cannot find a similar bug report in the repository, the *bug report master* should be contacted to make more advanced analysis and searches. For simplicity, from now, we will call it just as *submitter*.

Case study design. The submitter was instructed on how he should use both tools to search and analyze the bug reports. Thus, we divided the assessment period in two treatments: the first stage (from July 17 to August 07), the submitter should carry out the analysis first in the private organization's internal tool, and if he did not find a similar bug report, he should use our tool to perform a new analysis; in the second stage (from August 08 to August 29), this sequence was reversed.

We made the implementation of the study in this manner to reduce the factor of confusion regarding the knowledge of already analyzed bug reports. For example, if we chose to conduct the study so that the submitter performed the analysis in both tools, even if there was finding a similar bug report in the first tool used, the analysis in the second tool would be compromised because the submitter would already have knowledge about the existing similar bug report. In contrast, in a way that the study was implemented, we cannot know how much time it would be spent for the same analysis on both tools; this would only be possible if we performed another case study with two or more submitters.

Evaluation Metrics. To analyze which tool was more efficient with regard to the objectives defined, the following metrics were established:

- MT_1*: Type of bug reports analyzed.* With this metric, we want to characterize the

types of bug reports that were submitted during the course of the case study. For example, it is important to know the percentage of duplicate bug reports, valid bug reports, among others;

- *MT_2: Number of duplicate bug reports avoided.* With this metric, we want to know how many possible duplicate bug reports have been prevented with the use of our tool, and how many have been avoided with the use of the baseline tool;

- *MT_3: Time spent to analyze similar bug reports.* This metric is concerned with the time – in minutes – spent by submitters to search and analyze bug reports before submitting a new one. For this metric, we computed only the time for searches that found similar bug reports. In other words, we computed only the searches that avoided the submission of duplicates.

Quantitative analysis mechanisms. To analyze the data generated by the case study, it will be used descriptive statistics, such as percentages, mean, standard deviation and pie and bar charts.

Data gathering. For the metrics MT_1, MT_2 and MT_3, the submitter was responsible for recording the types of bug reports that were analyzed, the time spent on each analysis and, when a duplicate is found, to specify in which tool it was found. In the case of metric MT_1, if a bug report was not submitted and there were no similar bug reports for it, the submitter should specify the reason for does not submit it.

6.4 Result Analysis

During the case study, it were examined 144 bug reports by the *bug report master*. This amount leaded to 407 searches also performed by the *bug report master*, just in our tool during the study period. However, it was not possible to have access to the number of searches carried out in the private organization's internal tool due to issues of confidentiality. Following we analyzed each treatment of the case study design, and an analysis of the whole period (the two treatments together) are performed at the end. First of all, we provided a description of the bug reports analyzed in the study:

- **Unreproducible.** Errors that the testers and developers have been unable to reproduce;

- **Feature Not Yet Available.** Bug reports that describe errors that can only be corrected with the addition of new features;

- **WAD.** The term WAD is an acronym for *Work as Design*, and means that the error reported, in fact, is a behavior already expected in the software;
- **H/W Issue.** This type means that the error found is caused by a defect in the hardware, and it must not be submitted to the development team;
- **Workaround.** Bug reports of this type mean that the errors found can be avoided if the tests run in another way;

6.4.1 Analysis of the First Treatment

During the first treatment, it was analyzed 42 bug reports. It was the treatment in which less bug reports were analyzed, due to the fact that it was the beginning of the project being developed at the private organization. Figure 6.1 shows a graph with the classification of bug reports after the analysis performed by the submitter.

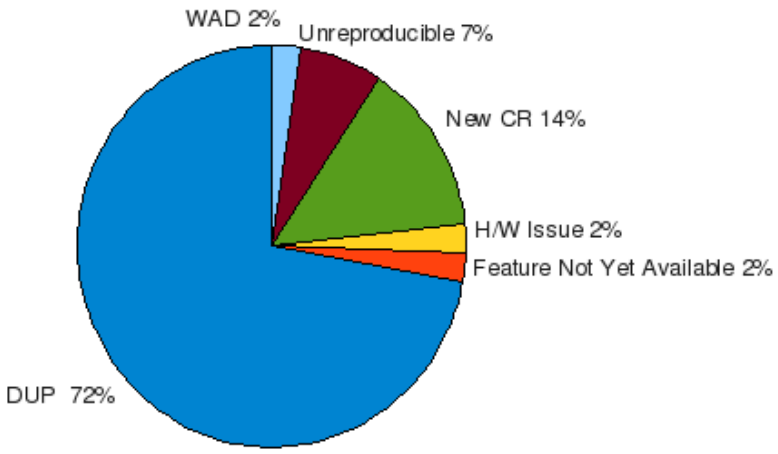


Figure 6.1 Repository status in the first treatment.

As it can be seen, 72% of bug reports that were analyzed would be duplicate if they were not avoided. Meanwhile, only 14% were valid bug reports, 7% were *Unreproducible*, 2% *WAD*, 2% *H/W Issue*, and 2% *Feature Not Yet Available*. These data show that the majority of time spent with bug report analysis and search is due to the presence of duplicates.

Furthermore, Figure 6.2 shows the percentage of duplicate bug reports that were avoided according to the tool used. In this first treatment, the bug reports were firstly analyzed using the baseline tool, and if it were not found similar bug reports, a new analysis should be performed with BAST. Thus, the analysis made with the baseline tool

prevented the submission of 58% of duplicates, while BAST prevented 35%. Moreover, 7% were avoided due to the information provided by developers, through email or other type of communication. Therefore, in this treatment, BAST had lower performance than the baseline tool in order to prevent duplicates.

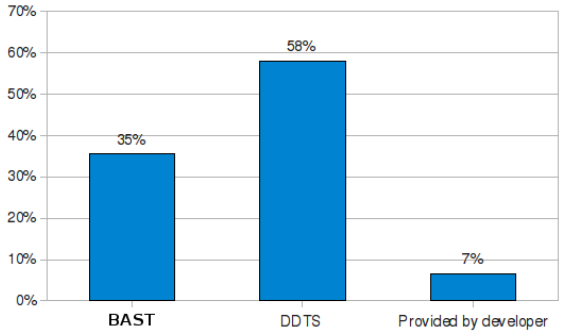


Figure 6.2 Duplicates found in the baseline tool and BAST in the first treatment.

The graph in Figure 6.3 shows the average time spent on search and analysis using the baseline tool and BAST. As can be seen, although BAST has avoided less duplicates than the baseline tool, as mentioned before, the time to do search and analysis with BAST was less than with the baseline tool. BAST saved almost the half the time that was spent with the baseline tool.

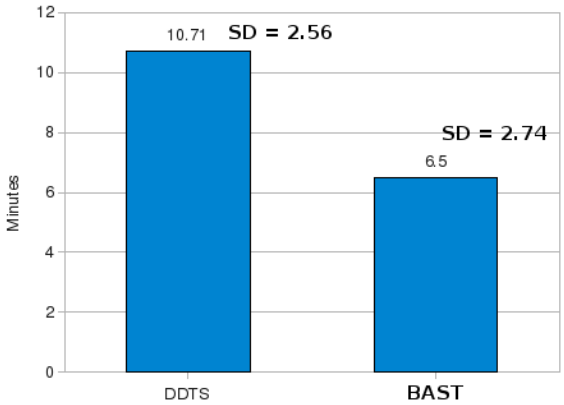


Figure 6.3 Time spent in searches for duplicates in first treatment.

6.4.2 Analysis of the Second Treatment

During the period of the second treatment, it was analyzed 99 bug reports. Note that the amount of analysis of bug reports has increased considerably in this treatment. One of the reasons for this growth was the maturing of the project under development. Figure

6.4 shows a graph with the classification of bug reports after the analysis conducted by the submitter during the second treatment. According to the chart, 44% of bug reports were duplicates that had been avoided, while 34% were valid bug reports, 9% were *unreproducible*, 7% *WAD*, 2% *H/W Issue*, 2% *workaround* and 1% *Feature Not Yet Available*.

Although the amount of bug report submitted increased, there was a reduction on the submission of duplicate bug reports and a growth of valid bug reports submission. We believe it is a consequence of the project maturation; testers and developers are more engaged and there is better knowledge of the bug reports currently being handled.

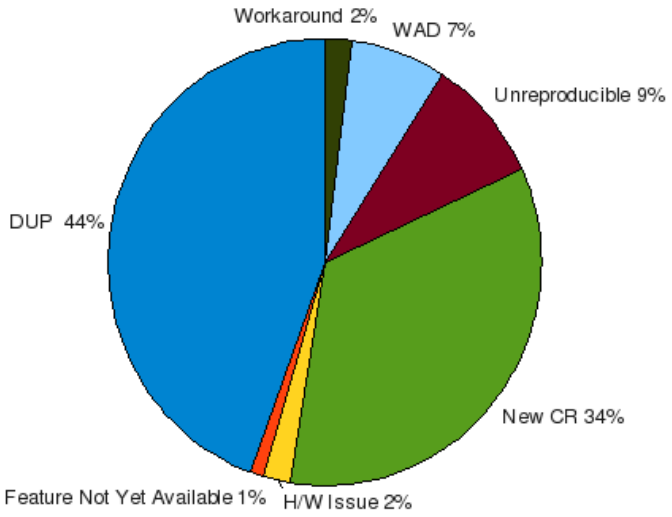


Figure 6.4 Repository status in the second treatment.

The graph in Figure 6.5 shows the percentage of duplicate bug reports that were avoided according to the tool used. In this second treatment, the bug reports were analyzed first using BAST, and if it was not found similar bug reports, a new analysis should be performed with the baseline tool. At the end of this treatment, the analysis made with BAST avoided the submission of 89% of duplicates, while the baseline tool prevented just 7%. In addition, 7% were avoided due to the fact that the submitter had prior knowledge of similar bug reports.

According to the graph in Figure 6.6, the average time spent on search and analysis of bug reports on both tools did not have big difference. Although BAST had a little better performance in the second treatment, in regard to the time of search and analysis, we believe that further analysis (i.e. to analyze the complexity of bug reports) should be done to decide whether such difference was caused by the use of BAST or if it was influenced by other factors.

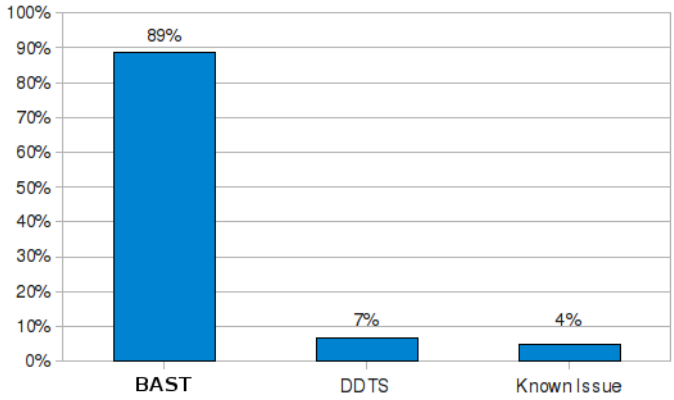


Figure 6.5 Duplicates found in the baseline tool and BAST in second treatment.

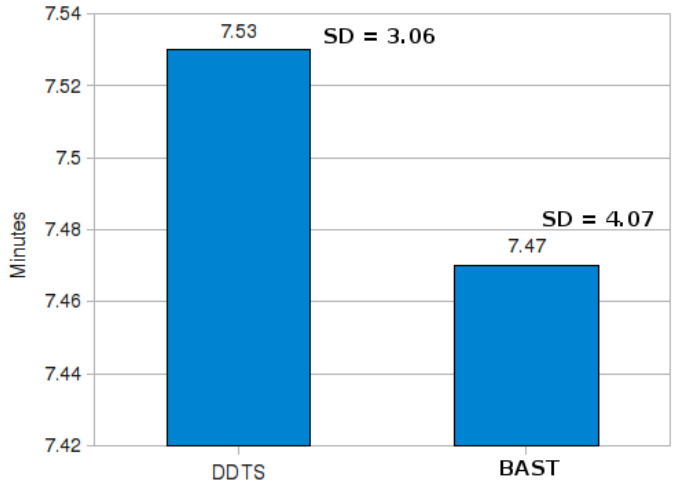


Figure 6.6 Time spent in searches for duplicates in second treatment.

6.4.3 Analysis of the Whole Period

In this subsection, we performed an analysis over the whole period of the case study, without distinguishing among the different treatments. Such analysis is important to understand the overall performance for both tools.

Types of bug reports analyzed. The graph in Figure 6.7 shows the types of bug reports that were analyzed by the submitter during the case study. As it can be seen, 53% was composed of duplicates, which would enter in the repository if it was not avoided. Meanwhile, only 29% of bug reports addressed issues not yet submitted, and thus were submitted to the repository. The other types of bug reports, although they were not duplicate, have not been submitted to the repository because they were invalid

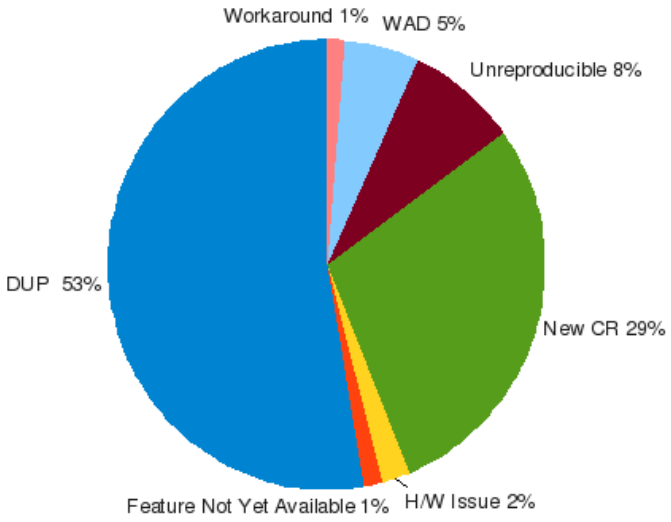


Figure 6.7 Repository status.

Thus, the graph of Figure 6.7 shows, once again, the severity of the problem of duplicate bug reports on projects of software development. For example, in this case study, if the 53% of duplicate bug reports were not prevented, the developers who were assigned to solve them would spend time, inevitably, with problems that have already been fixed or reported. That time could be used to solve other valid bug reports.

Percentage of duplicate bug reports avoided. As mentioned earlier, 53% of the bug reports were not submitted because they were duplicates. The graph in Figure 6.8 shows that 68% of the bug reports were avoided by using BAST, while only 27% have been avoided with the baseline tool. In addition, 3% of duplicates could only be avoided due to information provided by developers, or because the submitter just knew that such bug reports were duplicates.

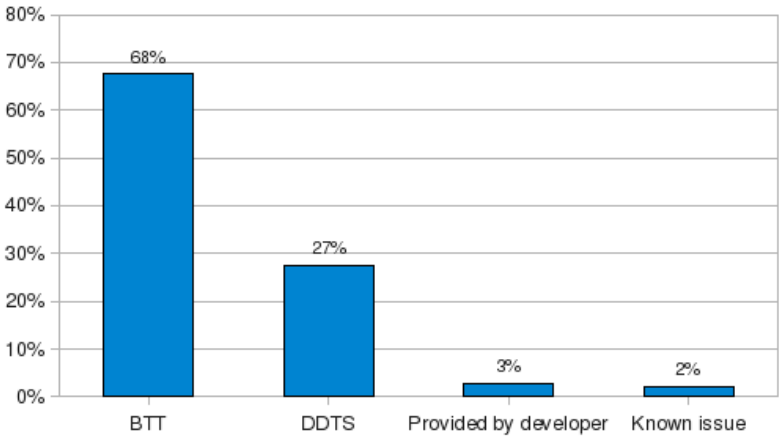


Figure 6.8 Duplicates found in the baseline tool and BAST.

Average time spent on analysis of bug reports. The graph in Figure 6.9 shows that, in general, the average time spent to perform analysis of bug reports is reduced when using BAST. On average, BAST saves half of time that would be spent if the submitter would be using the baseline tool. Therefore, it can be considered the the goal of reducing the time spent on analysis of bug reports was achieved.

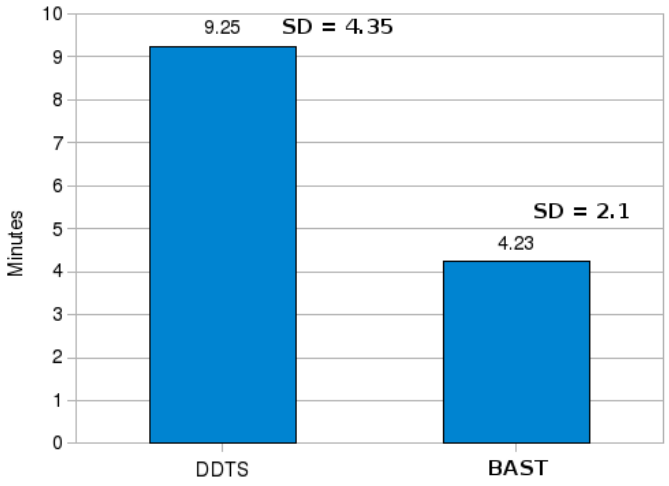


Figure 6.9 Time spent in searches for duplicates.

6.4.4 Analysis Conclusion

Although the individual analysis of the treatments showed few difference among the tools in some aspects, such as the time spent on analysis, according to the results analyzed by descriptive statistics of the whole period can confirm the alternative hypothesis μ_1 and

refute the null hypothesis μ_0. In other words, BAST can save more time than the baseline tool during the analysis of bug reports, and it can also avoid more duplicates.

6.5 Lessons Learned

In order to replicate this case study and increase its validity, some aspects must be taken into account. First of all, one must consider that the replication of this study is very dependent on an environment similar with the one presented in this work. Moreover, the conditions in which the case study was performed is very particular for the private organization, which makes difficult such replication.

Case study design. The way that the case study was designed can turn biased the measurement of which tool demands less time to perform search and analysis of bug reports. For example, the accommodation of the subject could lead him to prefers to use one tool instead of other. Thus, a better design for the case study would be to execute the search and analysis using only one tool in the first moment, and using the other tool in the second moment. However, this could not be done in this work due to environment constraints.

We must also consider that the time to perform a second analysis can be influenced by the first analysis. For example, if the submitter spends a certain time to perform an analysis in the first tool, but does not find any duplicate, it is likely that he will spend more time using the second tool to find a duplicate, if available. In this sense, we must also consider the factor of accommodation, in which the submitter tends to spend more time using the tool that he feels more comfortable to use.

Amount of bug reports in treatments. The amount of bug reports that were analyzed in each treatment were very different. However, this situation could not be controlled in the case study, because it was performed as an on-line case study, which means that the environment and conditions were real.

Lack of subjects. In addition, the number of subjects was not sufficient to generalize the case study results. It brings down the external validity of the study, at the same time that it does not enable the conclusion generalization even to the same organization. More subjects should be evaluated in order to increase the study validity.

Analysis mechanisms. We should use some hypothesis testing mechanisms, such as the stundent's t-test Wohlin *et al.* (2000), in order to analyze the time spent on analysis for each treatment. It is necessary because the difference between the values analyzed is very few. Thus, only looking at these values with descriptive statistics turns difficult to

draw a more concrete conclusion.

6.6 Summary

This chapter presented a case study to evaluate the BAST tool. The case study was conducted with only one subject inside a private organization from C.E.S.A.R. The BAST tool was compared against the baseline tool, and the results showed that the former can reduce the time to perform search and analysis of bug reports, at the same time it can increase the number of duplicate bug reports found. In addition, the confounding factors of the case study design and the lessons learned were also presented.

Next chapter presents another validation of BAST. It was performed an experiment with a set of students, where such students should use BAST and a baseline tool in order to compare the gains with BAST.

7

BAST Empirical Evaluation Experiment

Seven steps forward and you are not in the same place as six steps before.

—YGUARA (Chico's Thought Evolution)

7.1 Introduction

In previous chapters, we showed how duplicate bug reports is affecting software development, for example, with impact on total costs. We also described aspects of development teams and environments that can be possible causes for such problem. Moreover, it was described a tool developed in order to improve the analysis of bug reports and the duplicate detection.

The tool developed was evaluated in a private environment for software testing, with a real cycle of tests during, approximately, one moth. However, this case study was performed with only one tester, and we believe that such condition is a potential bias to the case study results, as we mentioned on the confound factors section. Thus, we performed a controlled experiment with 18 subjects with the objective to evaluate the tool against a baseline tool, so that more concrete conclusion can be drawn.

This chapter describes the experiment mentioned before, discussing its definition, planning, operation, analysis and interpretation, and other aspects concerning empirical experiments. The remaining of this chapter is organized as follows: Section 7.2 presents the definition of the experiment; in Section 7.3 it is presented the planning of the experiment; Section 7.4 describes how the the operation of the experiment was performed; in Section 7.5 the analysis and interpretation of the results are presented; and finally, Section 7.6 summarizes this chapter.

7.2 Definition

In order to define this experiment, it was used the GQM method (Basili *et al.*, 1986). With GQM, it is established the goal of the study, the questions to be answered, and the related metrics that must be collected to aid with the questions. However, the metrics are defined later as *independent variables*.

Goal. The goal of this experiment is to analyze **a tool to improve search and analysis of bug reports** for *evaluating it* with respect to its *effectiveness* and *efficiency* on detection of duplicate bug reports and time saving, in the *view point of* researchers, in the *context of* software development projects.

Questions. To achieve this goal, we defined *quantitative* and *qualitative* questions. The first ones are related to the data collected during the period that the experiment will be executed, and the last ones concerned with the submitters' feedback about the new approach adoption. The questions are described as follow.

Q_1. *Is there a reduction on the number of duplicated bug reports with the new tool adoption?* The main objective of the tool is to reduce the number of duplicates submission, thus, we would like to understand if this goal will be achieved.

Q_2. *Is there a reduction on the time that submitters spend to perform the search and analysis of bug reports with the tool adoption?* Another objective of the tool is to reduce the time needed to search and analysis of bug reports. Thus, it is important to know if time will be saved in such tasks.

Q_3. *Did the submitters have difficulties to use the tool?* In order to understand the difficulties that the submitters will face during the adoption of the tool, they will be asked to describe the difficulties faced during the experiment.

Object of study. The object of study is the tools used to analyze and search bug reports and their ability in terms of performance to save time and avoid duplicate bug reports.

Quality focus. The quality focus is the effectiveness and efficiency of the tool developed to aid the analysis and search activities. We want to understand if our tool can bring benefits to bug reports analysis and search.

Context. This experiment is concerned with the adoption of a tool developed to aid the bug report tracking process, focusing on search and analysis of bug report to avoid duplicates. The tool will be compared with a baseline tool. Thus, the subjects will use

both tools and it will be analyzed the time saved and duplicate bug reports avoided by both tools.

The experiment will be performed as an *off-line experiment*. The subjects will be composed by M.Sc. *students* from the Computer Science department at Federal University of Pernambuco/Brazil. In addition, the experiment will be performed distributed, which means that the subjects are free to choose their work environment, such as their home or university laboratories. Regarding the data used in the experiment, the subjects will use bug reports from some open-source projects.

7.3 Planning

Training. It will be explained to the subjects the context of the experiment and how it must be conducted. Furthermore, the subjects will be free to test the tools used in the experiment before the execution. The difficulties that may rise concerning the tools will be resolved through email.

Subjects selection. The subjects of this experiment will be selected by *convenience sampling* (Wohlin *et al.*, 2000; Kitchenham and Pfleeger, 2002). It means that the nearest and most convenient people from various elements of a population will be selected. For this experiment, it will be selected *twenty* subjects.

Instrumentation. All subjects will receive descriptions of defects from some open source software project. However, it is important to highlight that such descriptions are not the bug reports themselves, but only few words describing software errors. Thus, it will be created two lists of such objects for the experiment, where each list will contain 32 descriptions, being 50% with defects that already have bug reports describing them in the repository, and 50% with unique/not-reported defects. It is crucial that such list holds some descriptions about already submitted bug reports, thus we can determine which duplicates were correctly avoided.

The subjects will also receive guidelines to guide them through the experiment execution. In our experiment, the guidelines will be documents explaining how to perform the experiment and how to use the tools. Furthermore, the tools have their own manuals.

The results of the experiment (number of duplicates and time spent with search and analysis) will be collected using measurement instruments. Thus, it will be prepared *time-sheets* to collect the time spent by submitters with search and analysis of bug reports. Furthermore, questionnaires will be elaborated to gather qualitative data from subjects.

The instruments are presented in Appendix A.

Null hypothesis. The null hypothesis determines that there is no benefit of using BAST to perform analysis and search of bug reports. It is, there is no difference between the time saved and bug reports avoided by BAST and by baseline tool. The null hypothesis is specified as follows:

$$H_0: \quad \mu_{time\ with\ BAST} >= \mu_{time\ with\ baseline}$$
$$\mu_{duplicates\ avoided\ with\ BAST} <= \mu_{duplicates\ avoided\ with\ baseline}$$

Alternative hypothesis. The *alternative hypothesis* determines that BAST can save more time and avoid more duplicate bug reports than the baseline tool. Thus, there is difference between the time saved and bug reports avoided by BAST and by the baseline tool. The alternative hypothesis is specified as follow:

$$H_1: \quad \mu_{time\ with\ BAST} < \mu_{time\ with\ baseline}$$
$$\mu_{duplicates\ avoided\ with\ BAST} > \mu_{duplicates\ avoided\ with\ baseline}$$

Independent variables. This experiment has only one independent variable, which is the tool used to perform searches and analysis of bug reports in order to avoid duplicates.

Dependent variables. The dependent variables for this experiment are the (a) amount of *duplicate bug reports* and (b) the *time spent* with search and analysis of bug reports. The value for first variable is the percentage of duplicate bug reports present in a bug repository. The value for the second is the average time in *minutes* used to analyze bug reports.

Quantitative analysis mechanisms. Descriptive statistics will be used to analyze the data from the experiment. In order to test the hypotheses defined for the experiment, it will be used a paired *test-t* (Wohlin *et al.*, 2000). The *test-t* is a parametric mechanism for statistical analysis widely used when it is needed to compare the values from two treatments.

Qualitative analysis. The qualitative analysis will be conducted to understand the subjective aspects of the tool, such as the difficulties faced by the subjects during the use of the tool. The questionnaire presented in Appendix A.3 will be used to characterize these issues. The subject will be asked about the applicability and usability of the tool.

Blocking. Blocking is used to systematically eliminate the undesired effect in the comparison among treatments (Wohlin *et al.*, 2000). Initially, we are looking for subjects with similar experiences, in order to eliminate such undesired effects. However, only after collecting data about education and background, we will be able to determine if blocking is needed. The questionnaire in Appendix A.2 will be used to gather such information.

Randomization. The randomization applies on the allocation of the objects, subjects and in which order the tests are performed (Wohlin *et al.*, 2000). Since all subjects will participate in both treatments, no randomization is required.

Balancing. Since each treatment has the same number of subjects, the experiment is balanced, which strengthens the statistical analysis of the data (Wohlin *et al.*, 2000).

Experiment design. It will be used the *one factor with two treatments* design (Wohlin *et al.*, 2000) to perform the experiment. In such case, the factor is BAST. Thus, there is a treatment with such tool and another with the baseline tool. The two treatments are described as follow:

- **Treatment 1.** In the first treatment, the group of subjects will receive the list of error descriptions. Then, before submitting such errors as new bug reports, they will use the BAST tool from the selected project to perform searches and analysis of existing bug reports in order to avoid duplicates submission.

- **Treatment 2.** In the second treatment, the group of subjects will use the baseline tool to perform the searches and analysis of bug reports to avoid duplicates, as mentioned in Treatment 1. Furthermore, the group will use another list of error descriptions, thus the experiment will not be compromised by previous knowledge of error descriptions.

Figure 7.1 shows how the experiment execution is performed according to the treatments defined.

Internal Validity. The internal validity of the study is defined as the capacity of a new study to repeat the behavior of the current study, with the same subjects and objects with which it was executed (Wohlin *et al.*, 2000). The internal validity of the study depends on the number and expertise of the subjects. This study is supposed to have at least twenty subjects with similar background on bug report tracking activities to guarantee a good internal validity.

External Validity. According to (Wohlin *et al.*, 2000), the external validity is concerned with the generalization of the experiment results. Due to the time constraints, it will not be possible to apply the study in others research groups. Nevertheless, the external validity of the study is considered sufficient, since it aims to analyze the adoption of a tool for search and analysis of bug reports compared to a baseline. Thus, additional studies can be planned within projects that use bug tracker systems, considering the same profiles of subjects. Furthermore, all instruments used in the experiment will be available for download in `http://www.cin.ufpe.br/~ycc/btt-experiment`.

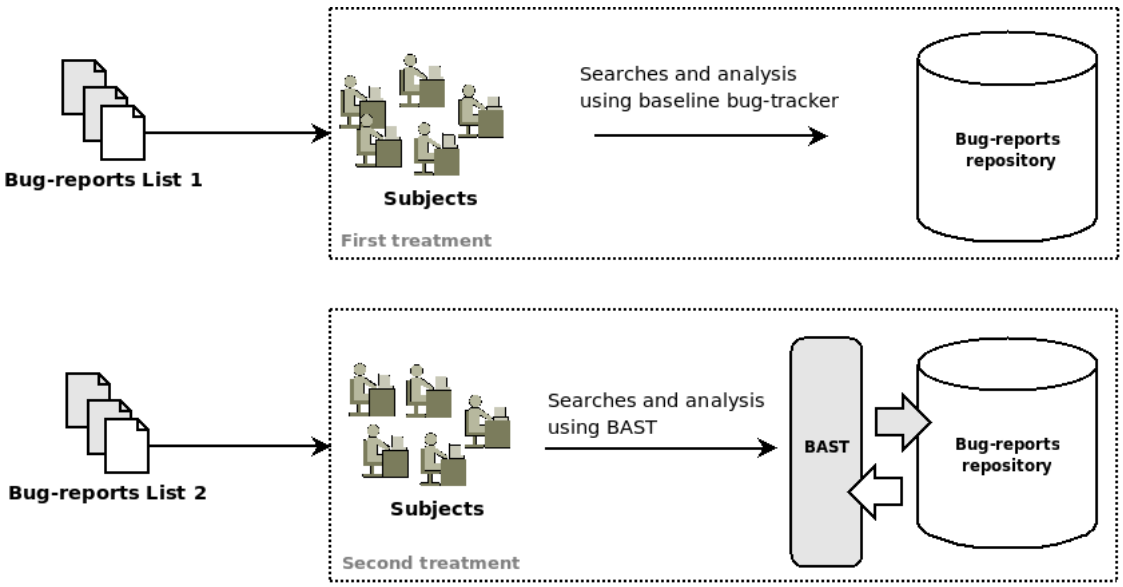


Figure 7.1 Experiment design. The subjects receive a list of error descriptions and perform searches and analysis using the baseline tool before submitting a new bug report. After that, the subjects receive another list, however they use BAST to perform searches and analysis.

Conclusion Validity. This validation determines the capability of the study to generate conclusions (Wohlin *et al.*, 2000). The analysis and interpretation of the results of this experiment will be described using descriptive statistics and statistical hypothesis testing, which are appropriate to the data type collected during the experiment.

Validity Threats. The following validity threats related to this experimental study were identified:

- **Boredom.** The amount of bug reports that the subjects have to consider is relatively large, being a work somewhat repetitive. Thus, some subjects may feel upset or disappointed with the experiment. In order to reduce the boredom of the experiment, the subjects will have a deadline of 5 days to execute each treatment;

- **Lack of Historical Data.** Although some work in the literature had performed some tests with computational techniques to detect duplicates, none of them has made an experiment involving people, as is the case of this experiment, in a such way we could compare our results with previous ones. Furthermore, we did not use the data from the case study presented in Chapter 6 because of the lack of subjects.

- **Environment.** The subjects will be free to do the tasks for the experiment in a place that suits them best. Thus, we believe that different environments can have positive or negative influences for the correct execution of the experiment. For

example, certain types of environments can stimulate the concentration of subjects, while other environments can stress them, thereby undermining the execution of the experiment.

- **Subjects Knowledge on bug reports.** The results of the experiment may be compromised by the lack or excess of knowledge on part of subjects on the bug reports of the selected project. Thus, if the subjects have any prior knowledge of errors already reported the analysis can be carried out faster.

- **Errors re-descriptions and fictitious errors.** The descriptions of errors, based on bug reports on the project to be selected, must be rewritten before being sent to the participants. This is necessary because if we maintain the same descriptions, the identification of the errors would be facilitated. However, the rewriting of the descriptions could change the meaning of the original description, making more difficult the bug report identification.

  Another threat are the fictitious errors that will be created to serve as the descriptions of unique errors. These descriptions may coincide with errors that have been reported to the project that will be selected and we are not aware of them, thereby increasing the number of duplicates on the list.

- **Halo Effect.** In a study conducted by Thorndike (1920), it was found a high correlation between ratings of logically unrelated traits from officer's assessments of their soldiers. It was because the ratings of each trait of a soldier was influenced by the officer's tendency to believe in their soldiers as generally good or bad. Porting the idea of *halo effect* to our experiment, some aspects (bad or good) from the tools can influence the evaluation of them by the subjects. For example, a poor user interface can influence the subjects and lead to negative evaluation of all other characteristics of the tool.

- **Internet Connection Constraints.** Both tools used in the experiment will run in the Web, thus the time to search for bug reports might be compromised by the quality of Internet connection from the subjects environment. Thus, since we do not differentiate the time to search and the time to perform analysis, the results can be imprecise.

7.4 Operation

The open source project. The open source project selected for the execution of the experiment was the Firefox Internet browser. We chose this project because it is a tool well known both by end users and Computer Science students. Being such a tool of broad knowledge, we believe that the subjects would have the minimum of trouble to understand the error descriptions.

Baseline tool. The Bugzilla bug tracker with all bug reports from Firefox was chose to compose the baseline. Since Firefox uses Bugzilla to handle its bug reports, it was convenient to choose this bug tracker. Furthermore, Bugzilla is one of the most known bug trackers, being used by several open source and private projects.

Selected bug reports. It was delivered to the participants of the experiment two lists, each one with 32 descriptions of error of Firefox. To compose this amount, for each list it was randomly chose 16 (50% of the list) bug reports that were submitted to the Firefox project during the year 2008, and the other 50% were composed of unique bug reports. In order to have unique bug reports not already reported to the Firefox bug tracker, it was created fictitious errors about Firefox features. Furthermore, such fictitious errors were based on another Internet browser, called Epiphany.

The subjects did not know what errors were present in the bug tracker of Firefox and which were fictitious. Moreover, the subjects were not informed that the unique errors were fictitious errors, so we ask them to not submit to the Firefox bug tracker the errors that were identified as unique during the analysis. The lists of bug reports can be seen in the Appendix B.

Subjects selection. We chose the subjects by convenient sampling method (Wohlin *et al.*, 2000; Kitchenham and Pfleeger, 2002). Thus, it was selected people from RiSE group to compose the subjects of the experiment. Such subjects are aware of the duplication problem, however it does not mean they care about the problem. Table 7.1 shows the profile of each subject. Only one of them is a Ph.D student (student #8), the others are M.Sc. students or have finished the M.Sc. course recently.

Training. Since all subjects have some experience with bug trackers, no training about how to use such tools was performed. However, information about the experiment execution were detailed, via email, to the subjects.

Cost. The subjects were suggested to perform the experiment activities on their free time, using the place more convenient for them. Furthermore, we needed only to setup the environment on Web for the BAST tool. The environment for baseline tool was already

ID	**Years since graduation**	**Participation in projects**	**# of projects with bug-tracker**	**Used bug-trackers**
1	3	15	4	Mantis
2	2	9	1	Trac
3	3	6	1	Mantis
4	1	13	3	Bugzilla, Trac
5	3	9	1	Mantis
6	1	6	3	Bugzilla, Mantis, Jyra, Other
7	2	6	4	Bugzilla, Trac, Mantis, Jyra
8	3	22	22	Bugzilla, Mantis, Other
9	0.5	4	0	–
10	3	4	4	Bugzilla, Mantis, Other
11	2.5	4	1	Bugzilla, Other
12	0.5	7	7	Trac, Mantis, Jyra
13	1	6	0	–
14	0.5	5	1	Bugzilla
15	1	1	1	Bugzilla, Trac, Other
16	5	16	16	Other
17	0.5	5	1	Bugzilla, Mantis
18	0.5	6	0	–

Table 7.1 Subjects profile.

done, since subjects used it directly from Firefox project.

7.5 Analysis and Interpretation

Table 7.2 shows the data obtained during the experiment. The first column shows the ID of the participant, second and third columns show the time spent in analysis of bug reports, and the remaining columns show the amount of duplicate bug reports avoided. As a first step in analyzing the data, it will be used descriptive statistics to visualize the data collected.

7.5.1 Quantitative analysis

Descriptive statistics Table 7.3 shows some statistics about the experiment results. For time spent on analysis, BAST had mean value of 4.54 minutes and standard deviation (SD) of 1.49, while Bugzilla had mean value of 4.32 and SD of 1.91. The differences among these values are very few, thus we plot boxes to better understand them, see Figure

	Time spent on analysis		Duplicates avoided	
#	BAST	Bugzilla	BAST	Bugzilla
1	3.09	2.56	5	9
2	3.03	2.47	8	10
3	3.31	3.09	8	12
4	6.78	6.84	13	10
5	5.1	4.82	4	2
6	3.06	2.75	11	11
7	4.97	3.91	12	9
8	5.04	9.56	2	8
9	5	2.97	8	8
10	3.63	3	7	10
11	6.84	6.88	7	8
12	1.78	2.66	6	4
13	6.66	5.41	9	10
14	3.69	4.19	13	10
15	6.47	4.31	9	11
16	3.75	2.72	8	10
17	4.47	4.91	6	8
18	4.97	4.78	0	0

Table 7.2 Collected data during the experiment.

7.2.

From Figure 7.2, we can conclude that people using Bugzilla to analyze bug report spent a little less time than using BAST. Figure 7.2 also shows an outlier for Bugzilla, however, we chose to keep it because the different between the outlier and the highest value is not so large. Furthermore, it is important to note that most of subjects from BAST keep their time spent on analysis below the median value. It is a positive point to BAST because we have more people spending less time (time bellow the median time) than Bugzilla.

For duplicate bug reports avoided, BAST had mean value of 7.56 bug reports avoided

	Time spent on analysis		Bug-reports avoided	
	BAST	Bugzilla	BAST	Bugzilla
Mean	4.54	4.32	7.56	8.33
Maximum	6.84	9.56	13	12
Minimum	1.78	2.47	0	0
SD	1.49	1.91	3.5	3.2

Table 7.3 Descriptive statistics.

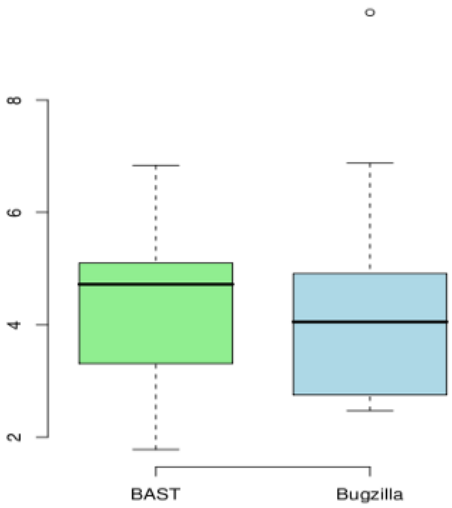


Figure 7.2 Box plot for time spent on analysis for both tools.

and SD of 3.5, while Bugzilla had mean value of 8.33 and SD of 3.2. Once again, we need to plot boxes to understand the differences among these values, see Figure 7.3. From Figure 7.3, we can conclude that Bugzilla helped to avoid more duplicates than BAST. Although Bugzilla is a little better than BAST concerning this aspect, we can observe that more subjects when using BAST can find higher number of duplicates than Bugzilla. Furthermore, we chose to keep the outliers from both boxes since they are in the expected bounds.

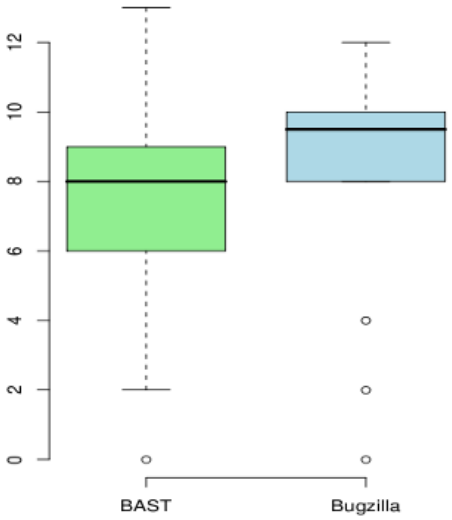


Figure 7.3 Box plot for duplicates avoided with both tools.

The descriptive analysis showed that Bugzilla had better performance than BAST in both cases: time spent on analysis and duplicates avoided. However, it is important to mention that the highest number of duplicate bug reports avoided were achieved with BAST and the minimum time spent on analysis too (see Table 7.3).

T-test. The data collected during the experiment were submitted to the *t-test* with 95% of confidence. Table 7.4 summarizes the results of the test. In the analysis of the time saved with both tools, the *t-test* did not reject the null hypothesis. Thus, we can conclude that there was no gain using BAST instead of Bugzilla to save time during the analysis of bug reports. In the analysis of duplicates avoided, the *t-test* also did not reject the null hypothesis, concluding that there is no advantage in using BAST to avoid duplicate bug reports.

	Time spent on analysis	**Duplicates avoided**
t_0	0.6292	-1.2466
Degrees of freedom	17	17
p-value	0.5376	0.2294
T distribution	2.11	2.11
Result ($t_0 > T$)	H_0: not rejected	H_0: not rejected

Table 7.4 T-tests applied with 95% of confidence to collected data.

Analysis of dependency. Table 7.5 shows a matrix of correlations for the aspects of subjects profile showed in Table 7.1, here used as independent variables for the correlations, and the results obtained for the dependent variables of the experiment. The correlations can variate from -1 to $+1$, and 0 (zero) means no correlation among the variables.

	BAST time	**Bugzilla time**	**BAST duplicates avoided**	**Bugzilla duplicates avoided**
Years of expereince	-0.132974628	-0.02296878	-0.19721612	0.183007854
Number of projects	-0.113791496	0.3720158	-0.28693026	-0.020936575
Bug-trackers used	-0.167706238	0.3511591	-0.26207382	0.052027136

Table 7.5 Matrix of correlation.

As we can observe, there is no correlations among the characteristics of subjects profile and the time spent to analyze bug reports and amount of duplicates avoided. It means, for example, that an experienced subject does not necessarily will consume less time on analysis than a less experienced subject, or will avoid more duplicates. The Appendix C shows the graphics for all possible correlation analysis.

7.5.2 Qualitative analysis

The result of the qualitative analysis about the BAST usability and functionality is summarized as follows. Such analysis was based on the questionnaire presented in Appendix A.3.

BAST features. From all subjects, seven (7) used the *filter features* provided by the tool, while eleven (11) did not use it. From those that used the filters, all of them told that it was useful for the analysis of bug reports. Furthermore, three (3) subjects told that would be interesting if the tool also provided other types of filters, such as: "*combination of keywords where you can specify mandatory and optional ones*", and "*query using some search operators such as AND, OR, +, -*". Seven (7) subjects told that the *ordering features* (to order the results of searching) are useful, while eleven (11) participants did not use them.

BAST Usability. From the seven (7) participants that used the filters, only one mentioned some difficult to use it, and only one subject from those that used the ordering features had problem with it. Moreover, some subjects mentioned that would be nice if BAST could visualize the pictures that are attached to the bug reports and to keep a history of the searches performed.

Four (4) subjects experienced problems with the visualization of bug reports details. These problems appeared due to issues from the infra-structure where BAST was hosted. In addition, some claimed about the time response for searches, however it happened also because of the host.

BAST usefulness. Fifteen (15) subjects believe the way bug report details are presented in BAST is more useful for the analysis than Bugzilla. Among their justifications, it was told that the user interface if very intuitive; it is possible to seen bug report details without leaving the list of search results. One subject wrote "*in fact, the way details are presented saves time to check them, since it is not necessary to open extra tabs or windows to see the details*", and other wrote "*it became easier to identify the duplicate bug reports and navigate among the details of them*".

7.5.3 Lessons Learned

In order to replicate this experiment, it is important to observe some aspects that could not be identified along the process of experimentation.

Training. The subjects were trained just by a document explaining how to perform the experiment and the basic features of the tool. Furthermore, the tool provides a help

that explains all features of it, however, only half of the subjects told that they used the help. Such lack of training was observed by the doubts that appeared during the execution. Thus, we believe that a better training about the tool and the experiment, in class room for example, should be performed before the experiment execution.

Infra-structure. BAST was hosted using the infra-structure from the Federal University of Pernambuco/Brazil, while Bugzilla was used from its own infra-structure on Mozilla host. Therefore, many subjects complained about the delay of search response while using BAST. We believe that it would be better if both tools were hosted using the same infra-structure, thus the delay of search response can be controlled.

Experiment design. One of the most important things in a experiment is its design. In our experiment, we chose to compare the same subjects using both tools, thus a single subject would be analyzed using BAST and after using Bugzilla. We believe that such design affected the performance of BAST because of subject's boredom during the experiment execution. We observed that seventeen (17) performed the experiment in the same day, thus it is possible that the level of boredom was high while using Bugzilla (the second tool in sequence). Thus, it would be better if one replicating this experiment could use two groups of subjects, each of them using a different tool.

Experiment instruments. We observed that some bug reports were found independently of the tool used by all subjects, which let us to conclude that these bug reports have keywords in their content that become them easy to be found. Thus, for an experiment replication, the bug reports must be more carefully selected and validated.

Analysis. One important thing that is missing in this experiment, and must be performed in replications of it, is the the analysis of correlation among the subjects profile and the collected data from the experiment. For example, it is important to analyze if more experienced subjects spend less time than less experienced ones, or if more experienced subjects can avoid more duplicate bug reports than less experienced ones.

7.5.4 Conclusion

The descriptive statistics showed that Bugzilla had a little better performance for both aspects studied: time spent on analysis and amount of duplicate bug reports avoided. However, the differences among the data analyzed for both tools in the descriptive statistics were not significant, which turns hard to draw a concrete conclusion saying what tool is better.

Furthermore, the t-test applied to test the hypotheses did not provide sufficient data to reject the null hypothesis. However, the qualitative analysis showed that BAST have

many good aspects that should be taken into account before choosing one of the tools. It was clear that subjects felt more comfortable while using BAST than Bugzilla due to its usability.

Finally, more experiments should be performed taking into account the lessons learned before. We believe that another experiment following these issues can be more conclusive saying which tool is better to save time on analysis of bug reports or to avoid duplicates.

7.6 Summary

This chapter presented the definition, planning, operation, analysis and interpretation of the experiment performed to evaluate the tool developed in this work, BAST. The experiment was conducted with 18 subjects, and the tool was compared with a baseline tool, which in this case was the Bugzilla bug tracker.

The results of the experiment pointed out that the difference between using BAST or using Bugzilla for analysis and search of bug reports, in terms of reducing time and avoiding duplicates, is very few. However, qualitative analysis showed that developers prefer to perform such tasks using BAST, due mainly to its usability.

Furthermore, it is clear in the experiment that replications of this experiment must be performed, taking into account the lessons learned, in order to draw more concrete conclusions.

Next chapter presents the concluding remarks and future work of this dissertation.

8

Concluding Remarks and Future Work

Eight steps forward and you are not in the same place as seven steps before, ad infinitum.

—YGUARA (Chico's Thought Evolution)

This work proposed and evaluated a solution to the bug reports duplication problem. As it was described, this problem is present in all the projects investigated, and the problem is characterized by the submission of two or more bug reports that describe the same software change/issue. The main consequence of this problem is the overhead of rework when managing these bug reports. In other words, people involved with bug report analysis, inevitably, spend time with search and analysis of existing bug reports, to ensure that duplicates will not be submitted.

Thus, it was presented the state-of-the-art in mining bug report repositories, detailing the work that have addressed the problem of bug report duplication, and describing some work concerning other issues, mentioned before, about bug trackers. A characterization of the duplication problem was also performed, where it was observed several projects (private and open source projects) in order to understand the potential causes and consequences of duplication problem.

Given the importance and severity of the problem, and the constant search for cutting costs in software development organizations, it was developed BAST, a tool for search and analysis of bug reports. The tool was evaluated twice. The first evaluation was performed with only one subject inside a private organization for software development and tests. In that case, the results showed that BAST have better performance than the baseline tool.

In the second evaluation, the tool was used by 18 subjects in an academic environment.

Also in this evaluation, we compared BAST against a baseline tool. The quantitative analysis of the experiment did not show many differences between both tools, however qualitative analysis indicated that BAST is a better choice to perform analysis and search of bug reports due to its usability.

8.1 Research Contribution

This work has five main contribution: 1) it presented a characterization of the state of the art on mining bug repositories; 2) it was presented a fully detailed characterization of the bug report duplication problem, involving several projects; 3) a tool, called BAST, was proposed to reduce the time spent with search and analysis of bug reports (due to duplication problem); 4) a case study was conducted inside a private organization to evaluate the tool proposed; 5) in addition, it was performed an experiment with 18 subjects to evaluate the proposed tool against a baseline tool.

State-of-the-art. We described several work in literature that had investigated bug repositories, discussing the solutions that approached the bug report duplication problem, and briefly described some work that approached bug repositories for other purposes.

Characterization study of the problem. The objective was to investigate if duplication problem is a real problem and, if so, what is the impact on software development and which characteristics from software projects could be potential factors to the problem.

BAST. Based on current state-of-the-art in approaches to solve the duplication problem, and characteristics discovered in the characterization study, it was developed a tool to facilitate the search and analysis of bug reports.

Case study. After the tool development, it was performed a case study inside a private organization for software tests. This case study was held with only one subject, and the objective was to evaluate the BAST tool against a baseline tool.

Experiment. With this experiment the objective was to evaluate the BAST tool with more subjects (18 subjects) against a baseline tool. The experiment was defined and planned according to Wohlin *et al.* (2000), and the results were presented using descriptive statistics and hypotheses testing.

Final product. The BAST tool is an initial prototype. It must be evolved, including other techniques and features, and it is a candidate to be commercialized with other tools offered by RiSE[1]. In addition, the BAST tool is still being used in the private organization where the case study was held; according to the users, the tool is facilitating their work.

[1] http://www.rise.com.br

8.2 Future Work

In this work it was developed and evaluated an initial prototype. Thus, we are aware that some enhancements and features must be implemented, also as some defects must be fixed. Some of the enhancements and defects were reported by users, and others were left out because they were out of scope of the master degree. Thus, some important aspects are described as follows:

Evolve from prototype. The prototype must be evolved in order to be a commercial application. Thus, it must be developed interface improvements, usability concepts, security constraints, among others.

Information visualization. Also concerning the prototype evolution, techniques for information visualization (Card *et al.*, 1999) are being studied in order to be included in the tool. This techniques will help users to perform the analysis of bug reports through visualization.

Alternative integration methods. Currently, the BAST tool enables the users to import bug reports from bug repositories through zip files, or integrating the database from bug repositories directly to the BAST database. Other methods, such as the use of web services, must be investigated and implemented in order to facilitate the integration of BAST to existing bug repositories.

Search and raking techniques. We believe that modern techniques to facilitate searches can be incorporated to the BAST to improve search features. For example, it can be added search through tags and query reformulation (Baeza-Yates and Ribeiro-Neto, 1999). Furthermore, other ranking techniques could be useful to improve search results precision.

For example, including the number of comments on each bug report as a parameter of the ranking algorithm, can improve the search precision to find duplicates. This idea is based on the fact the the bug reports with high number of comments are popular issues, thus being more prone to receive duplicate bug reports.

Experiment replications. The experiment performed in this dissertation must be replicated taking into account the many lessons learned and threats observed during its execution. Although the qualitative analysis indicates that BAST is preferable to execute analysis and searches of bug reports, the quantitative analysis must be more conclusive, and this is possible by replicating the experiment.

Bibliography

Almeida, E. S., Alvaro, A., Lucrédio, D., Garcia, V. C., and Meira, S. R. L. (2004). Rise project: Towards a robust framework for software reuse. In *IEEE International Conference on Information Reuse and Integration (IRI)*, pages 48–53, Las Vegas, NV, USA.

Alvaro, A., Almeida, E. S., and Meira, S. L. (2006). A software component quality model: A preliminary evaluation. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, pages 28–37, Washington, DC, USA. IEEE Computer Society.

Antoniol, G., Penta, M. D., Gall, H., and Pinzger, M. (2005). Towards the integration of versioning systems, bug reports and source code meta-models. *Electronic Notes in Theoretical Computer Science*, **127**(3), 87–99.

Anvik, J. and Murphy, G. C. (2007). Determining implementation expertise from bug reports. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07)*. IEEE Computer Society.

Anvik, J., Hiew, L., and Murphy, G. C. (2005). Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, New York, NY, USA. ACM Press.

Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proceeding of the 28th International Conference on Software Engineering (ICSE'06)*, pages 361–370, New York, NY, USA. ACM Press.

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (1999). *Modern Information Retrieval*. ACM Press / Addison-Wesley.

Basili, V., Selby, R., and Hutchens, D. (1986). Experimentation in software engineering. *IEEE Transactions on Software Engineering*, **12**(7), 733–743.

Bennett, K. H. and Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)*, pages 73–87, New York, NY, USA. ACM Press.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2007). Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange (Eclipse '07)*, pages 21–25. ACM Press.

Brito, K. S. (2007). *LIFT: A Legacy InFormation retrieval Tool*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

Canfora, G. and Cerulo, L. (2005). Impact analysis by mining software and change request repositories. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, page 29, Washington, DC, USA. IEEE Computer Society.

Canfora, G. and Cerulo, L. (2006). Supporting change request assignment in open source development. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC'06)*, pages 1767–1772. ACM Press.

Card, S. K., Mackinlay, J. D., and Shneiderman, B. (1999). *Readings in Information Visualization: Using Vision to Think*. The Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann.

Cavalcanti, Y. C., Martins, A. C., Almeida, E. S., and Meira, S. R. L. (2008). Avoiding duplicate cr reports in open source software projects. In *The 9th International Free Software Forum (IFSF'08)*, Porto Alegre, Brazil.

Durao, F. A. (2008). *Semantic Layer Applied to a Source Code Search Engine*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

Eastwood, A. (1993). Firm fires shots at legacy systems. *Computing Canada*, **19**(2), 17.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, **2**(3), 17–23.

Feldman, R. and Sanger, J. (2007). *The Text Mining Handbook: advanced approaches in analyzing unstructured data*. Cambridge University Press.

Filho, E. D. S., Cavalcanti, R. O., Neiva, D. F. S., Oliveira, T. H. B., Lisboa, L. B., Almeida, E. S., and Meira, S. R. L. (2008). Evaluating domain design approaches using systematic review. In R. Morrison, D. Balasubramaniam, and K. E. Falkner, editors, *2nd European Conference on Software Architecture (ECSA'08)*, volume 5292 of *Lecture Notes in Computer Science*, pages 50–65. Springer.

Fischer, M., Pinzger, M., and Gall, H. (2003a). Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 90–99, Washington, DC, USA. IEEE Computer Society.

Fischer, M., Pinzger, M., and Gall, H. (2003b). Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM'03)*, pages 23–32. IEEE Computer Society.

Garcia, V. C., Lisboa, L. B., ao, F. A. D., Almeida, E. S., and Meira, S. R. L. (2008). A lightweight technology change management approach to facilitating reuse adoption. In *2nd Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS'08)*, Porto Alegre, Brazil.

Hiew, L. (2006). *Assisted Detection of Duplicate Bug Reports*. Master's thesis, The University of British Columbia.

Huff, F. (1990). Information systems maintenance. *The Business Quarterly*, (55), 30–32.

Jalbert, N. and Weimer, W. (2008). Automated duplicate detection for bug tracking systems. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, pages 52–61. IEEE Computer Science Press.

Kagdi, H., Maletic, J., and Sharif, B. (2007a). Mining software repositories for traceability links. In *Iin the Proceedings of the 15 IEEE International Conference on Program Comprehension (ICPC'07)*, pages 145–154.

Kagdi, H., Collard, M. L., and Maletic, J. I. (2007b). A survey and taxonomy of approaches for mining software repositories in the context of software evolution: Survey articles. *Journal of Software Maintenance and Evolution*, **19**(2), 77–131.

Kitchenham, B. and Pfleeger, S. L. (2002). Principles of survey research: part 5: populations and samples. *SIGSOFT Software Engineering Notes*, **27**(5), 17–20.

Ko, A. J., Myers, B. A., and Chau, D. H. (2006). A linguistic analysis of how people describe software problems. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC'06)*, pages 127–134, Washington, DC, USA. IEEE Computer Science.

Koponen, T. and Lintula, H. (2006). Are the changes induced by the defect reports in the open source software maintenance? In H. R. Arabnia and H. Reza, editors, *Proceedings of the 2006 International Conference on Software Engineering Research (SERP'06)*, pages 429–435. CSREA Press.

Koskinen, J. (2004). Software maintenance costs. http://www.cs.jyu.fi/~koskinen/smcosts.htm.

Lancaster, F. W. (1986). *Vocabulary Control for Information Retrieval*. Information Resources Press, 2 edition.

Lientz, B. P. and Swanson, E. B. (1981). Problems in application software maintenance. *Communications of the ACM*, **24**(11), 763–769.

Martins, A. C., Garcia, V. C., Almeida, E. S., and Meira, S. R. L. (2008). Enhancing components search in a reuse environment using discovered knowledge techniques. In *2nd Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS'08)*, Porto Alegre, Brazil.

Mascena, J. C. C. P. (2006). *ADMIRE: Asset Development Metric-based Integrated Reuse Environment*. Master's thesis, Federal Uniersity of Pernambuco, Recife, Pernambuco, Brazil.

McKee, J. R. (1984). Maintenance as a function of design. In *AFIPS National Conference Proceeding*, volume 53, pages 187–1983.

Mendes, R. C. (2008). *Search and Retrieval of Reusable Source Code using Faceted Classification Approach*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

Moad, J. (1990). Maintaining the competitive edge. *Datamation*, **4**(36), 61–62.

Nascimento, L. M. (2008). *Core Assets Development in SPL - Towards a Practical Approach for the Mobile Game Domain*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

Panjer, L. D. (2007). Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07)*, page 29, Washington, DC, USA. IEEE Computer Society.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B. (2003). Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 465–475, Washington, DC, USA. IEEE Computer Society.

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*.

Port, O. (1988). The software trap – automate or else. *Business Week*, **9**(3051), 142–154.

Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, **3**(14), 130–137.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *IEEE Computer*, **33**(10), 23–29.

Pressman, R. S. (2004). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science.

Python Software Foundation (2008). Python Programming Language. http://www.python.org. Last access on May/2008.

Runeson, P., Alexandersson, M., and Nyholm, O. (2007). Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 499–510. IEEE Computer Science Press.

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

Sandusky, R. J., Gasser, L., and Ripoche, G. (2004). Bug report networks: Varieties, strategies, and impacts in a f/oss development community. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR'04)*, pages 80–84, University of Waterloo, Waterloo.

Santos, E. C. R., ao, F. A. D., Martins, A. C., Mendes, R., Melo, C., Garcia, V. C., Almeida, E. S., and Meira, S. R. L. (2006). Towards an effective context-aware proactive asset search and retrieval tool. In *6th Workshop on Component-Based Development (WDBC'06)*, pages 105–112, Recife, Pernambuco, Brazil.

Shuja, A. and Krebs, J. (2007). *Ibm®rational unified process®reference and certification guide: solution designer*. IBM Press.

Sommerville, I. (2007). *Software Engineering*. Addison Wesley, 8 edition.

Song, Q., Shepperd, M. J., Cartwright, M., and Mair, C. (2006). Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, **32**(2), 69–82.

Succi, G. (2001). *Extreme programming examined*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Thorndike, E. L. (1920). A constant error in psychological ratings. *Journal of Applied Psychology*, (4), 25–29.

Vanderlei, T. A., ao, F. A. D., Martins, A. C., Garcia, V. C., Almeida, E. S., and Meira, S. R. L. (2007). A cooperative classification mechanism for search and retrieval software components. In *Proceedings of the 2007 ACM symposium on Applied computing (SAC'07)*, pages 866–871, New York, NY, USA. ACM.

Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J. (2008). An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 13th International Conference on Software Engineering (ICSE'08)*, pages 461–470. ACM Press.

Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07)*, pages 20–26. IEEE Computer Society.

Wohlin, C., Runeson, P., Martin Höst, M. C. O., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. The Kluwer Internation Series in Software Engineering. Kluwer Academic Publishers, Norwell, Massachusets, USA.

Zelkowitz, M. V., Shaw, A. C., and Gannon, J. D. (1979). *Principles of Software Engineering and Design*. Prentice Hall Professional Technical Reference.

Appendices

A

Experiment Instruments

A.1 Time sheet

ID	*Start date*	*Start time*	*End date*	*End time*	*Is it a duplicate*?
1	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
2	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
3	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
4	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
5	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
6	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
7	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
8	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
9	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
10	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
...					
23	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
24	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
25	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
26	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
27	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
28	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
29	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
30	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
31	/ /	:	/ /	:	( ) Yes. ( ) No. ID:
32	/ /	:	/ /	:	( ) Yes. ( ) No. ID:

Table A.1 Time sheet used in the study.

A.2 Questionnaire for Subjects Profile

Questionnaire for Subjects Profile
How many years since graduation? ( ) years.
How many projects do you have participated according to the following categories? ( ) Low complexity. ( ) Medium complexity. ( ) High complexity.
What were the roles that you played in the projects cited before (developer, configuration manager, tester...)?
How do you define your experience with bug-trackers? ( ) I never used them before. ( ) I used them in every project i participated. I used them in ( ) projects.
Do you have used any of the following bug-trackers? ( ) Bugzilla. In: ( ) industry ( ) academia ( ) Trac. In: ( ) industry ( ) academia ( ) Mantis. In: ( ) industry ( ) academia ( ) Jyra. In: ( ) industry ( ) academia ( ) BSD Bug-tracker. In: ( ) industry ( ) academia ( ) Other:
Have you performed any analysis of Firefox bug-reports before? ( ) Yes. ( ) No.

Table A.2 Questionnaire for bug-report submitters.

A.3 Form for Qualitative Analysis

Questionnaire for Qualitative Analysis
Did you use any of the search filters provided by BAST? ( ) Yes. ( ) No.
Is there any search filter you think it must be present in BAST? ( ) Yes. ( ) No. Cite them:
Did you have any problem with the search filters usage? ( ) Yes. ( ) No. Cite them:
Did you use the ordering features of BAST? ( ) Yes. ( ) No.
Did you have any problem with ordering features? ( ) Yes. ( ) No. Cite them:
Do you think there is any other important information that must be present in the list of search results? ( ) Yes. ( ) No. Cite them:
Did you have any problem to visualize the details from some bug-report? ( ) Yes. ( ) No. Cite them:
Do you believe the way bug-reports details are presented was helpful to perform the analysis? ( ) Yes. ( ) No.
Was the recommendation of related bug-reports, presented in the bug-report details, useful for the analysis? ( ) Yes. ( ) No.
Is there any other information concerning bug-reports details you believe it should be present or emphasized? ( ) Yes. ( ) No. Cite them:
Did you use the help provided by BAST? ( ) Yes. ( ) No.
Did you found any other problem/enhacement/defect that was not mentioned before? Cite them.
Please, write down any suggestion you think might would be useful.

Table A.3 Questionnaire for qualitative analysis.

B

Bug-reports Used in the Experiment

B.1 First List of Bug-reports Used in the Experiment

Table B.1 First list of bug-reports used in the experiment

ID	*Firefox ID*	**Summary/Description**	**Status**
Bug-reports present in Firefox bug-tracker			
1	437878	Add bookmark dialog Two folders/directories with same name not distinguishable in quick list	NEW
2	430901	The checkbox in the Tools>Options menu, under the Privacy tab, the "Keep my History for at least X days" will not save any changes, and remains unchecked.	FIXED
3	458981	cannot add bookmarks to bookmarks toolbar (tried clicking on the star and dragging a link to the toolbar) – NS_ERROR_FILE_CORRUPTED	FIXED
4	438252	Editing bookmark's uri and then Tags, update tags for the old uri	FIXED
5	411003	Pressing Escape key should cancel adding a bookmark	DUPLICATE
6	413140	Bookmarks are "sortable" in bookmark manager but show up as unsorted in drop down menu	DUPLICATE
7	411214	New MIME type set with "Do this automatically ..." fails to locate helper app	FIXED

Continued on next page

ID	*Firefox ID*	**Summary/Description**	**Status**
8	414735	New application details window spewing errors	FIXED
9	425419	Help window opened from the Options dialog is modal (Windows only)	FIXED
10	439133	“Show Image” is poorly labeled, can be confused with “View Image”	FIXED
11	429119	menu bar missing if Firefox is not default browser	FIXED
12	415232	Right-click context menu is broken ("Open Link in New Window" doesn’t work)	FIXED
13	462041	Refresh the Privacy preference pane	NEW
14	412256	touchpad scrolling still moves back/forward in history	NEW
15	413609	Can’t change the application used to launch a document type	NEW
16	442736	If browser.startup.page is equal to 3, browser.warnOnQuit should be changed to false	NEW
Fictitious error descriptions			
17		crash in Epiphany Web Browser: Seems to be avahi bookmarks stuff (Crashes a few seconds after launch). Repeatedly.	
18		ga_client_start() called twice	
19		crash in Web Bookmarks: Watching a video on youtube and reading boingboing. It get the problem at the final of the video (a problem in the flash plugin?)	
20		crash in Epiphany Web Browser: Such a crash just happens every single time I resume from hibernation and firefox was running before hibernating (obviously).	

Continued on next page

ID	*Firefox ID*	**Summary/Description**	**Status**
21		Possible data loss due to UI problem ("All" topic being rename-able). This of course is not good, it's even worse when I lost my bookmarks because I renamed the topic lots of time without noticing what was happening.	
22		Topics with commas not handled correctly in Add Bookmark dialog	
23		undo for bookmark deletion. Bookmarks accidentally deleted within a bookmark editing session should be recoverable.	
24		Can't add bookmarks by dragging them into the Bookmarks window	
25		smart bookmarks: Empty text field after each use. The text field of a smart bookmark in firefox should be cleared after use. That would make it faster to use a smart bookmark many times in a row, because the user doesn't have to spend time deleting prior searches.	
26		Hebrew bookmarks are not displayed. I'm using Firefox with an English locale, and I have bookmarks with both English and Hebrew titles. While the ones with the English titles are displayed properly in the bookmarks menu, the ones with the Hebrew titles appear as empty menu items (except for the favicon, for websites that have one).	
27		Bookmarks context menu. We can have bookmarks toolbar. Nice, but context menu doesn't allow to remove or move the bookmark.	
28		new imported bookmark topics will not show. When editing the bookmarks, new topics wont show in the bookmark menu. They do show in the bookmark edit window	

Continued on next page

ID	*Firefox ID*	**Summary/Description**	**Status**
29		Bookmarks menu freezes/crashes Firefox. I've enabled the extension to synchronize my bookmarks with my del.icio.us account. However I don't think the problem is there, unless it's importing them oddly.	
30		Firefox dont recognize all the favicon of the sites. i've noticed that some sites, that have a favicon, when saved in bookmarks don't have a favicon...	
31		Can not import from any HTML or RDF file, even if Firefox created the file. Firerox on Ubuntu Hardy cannot import bookmarks from either bookmarks.html or Bookmarks.rdf. The import fucntion always bombs out with a dialog stating that the file is probably not of the supported type.	
32		Firefox crashes when all topics are selected. I noticed that if you want to add a bookmark (press D̂) and you select "show all topics" and then check all the checkboxes - firefox crashes. The order is not important how you check them.	

Second List of Bug-reports Used in the Experiment

Table B.2 Second list of bug-reports used in the experiment

ID	*Firefox ID*	**Summary/Description**	**Status**
Bug-reports present in Firefox bug-tracker			
1	420085	First flash site is fine, second or third flash site the browser crashes	UNCO
2	421109	Firefox crashed on font change in gnome appearance preferences	UNCO
3	456609	Started immediately after upgrading, Cooliris (piclens) crashes when scrolling google images	UNCO
4	410388	Firefox produces a dialogue box which says firefox must close and then shuts down	UNCO
5	410389	List All Tabs control's contents isn't scrolled via touchpad	UNCO
6	410402	Mozilla take more time when open	UNCO
7	410416	images are not saved when trying to do so	UNCO
8	410464	Legends and Nested Legends will not have a width applied	UNCO
9	410468	Cannot access URLs with different characters from english	UNCO
10	410473	Outlook 2007 general failure when trying to open hyperlink in Firefox	UNCO
11	410481	Barcode for itinerary doesn't display, but does in IE	UNCO
12	410529	No print feedback any more	UNCO
13	410534	A PDF file should be put in cache and shouldn't be downloaded several times	UNCO
14	410549	No memories with Hotmail	UNCO
15	410589	Mozilla Firefox Keeps searching after ppage is complete	UNCO
16	410646	Eats my computer memory	UNCO
Fictitious error descriptions			
17		Firefox crash during closing. Firefox sometimes crash during closing the latest window of it.	

Continued on next page

ID	*Firefox ID*	**Summary/Description**	**Status**
18		Segfault while printing a page to a postscript file.	
19		Firefox crash during Java load.	
20		Firefox crashed when switching to full screen mode. Firefox just crashed when switching to full-screen mode.	
21		Crash at startup and reopening of tabs. I clicked a link in a program. This made Firefox start up, then it asked if it should reload the tabs from last time (I can't remember which webpages I was visiting at that time. It crashed some days ago, and I haven't used it voluntairily since). At the moment Firefox loaded the webpage it couldn't handle, it crashed again. It didn't show me which webpage this was, so it's hard for me to tell.	
22		Broken typeahed checker with xullrunner. I compiled xulrunner with –enable-extensions=-typeaheadfind, but config script errored on typeaheadfind check	
23		Session recovery creates duplicate windows. Mostly though not always, the session recovery duplicates the windows, so that there are two exactly identical sets of windows and tabs. I have no idea what causes that, maybe it's because I have quite a lot of items in the session_creashed.xml file.	
24		autocompletion lists from the location bar "drops" up. i type a few letter in the location bar. sometimes the list didnt drop down. it jumps up and i couldnt see proposals for completion. i tested a little bit.	
25		Status bar should be empty when there is no status. The status bar currently instructs people to "Enter a web address to open, or a phrase to search for" when it doesn't have anything better to say.	

Continued on next page

ID	*Firefox ID*	**Summary/Description**	**Status**
26		Right-click menu should show image filename. It does not provide an easy way to discover the filename of the image. One option is to copy the complete url to the clipboard and paste it into another application (lot of work). Another option is to hover over the "Open Image" option and watch the scrollbar.	
27		Default font size is not used in new users. In Firefox, sans serif fonts are used right from the start. However, most fonts are too small. Only after going to “Preferences” -> “Fonts & Style” -> “Detailed Font Settings...”, the size of the concerned fonts jumps instantaneously to the right values. This is apparently due to the same bug that caused the use of serif instead of sans serif fonts.	
28		If Location entry is not in the toolbar, there’s no reaction to control+L. The location bar does not show up after hitting Ctrl-L with all toolbars hidden.	
29		When starting, no URL is shown in the address bar. I was testing GNOME 2.13.91. When starting firefox on a fresh account, it opens Google as homepage. However, there’s nothing in the address bar, while it would make sense to show the google URL.	
30		Bookmarks export to HTML produces strange categorization. When exporting bookmarks to HTML, the generated HTML contains random (at least to m) headers. I think the current system is just not suited to the new bookmarks layout. What about rendering the same structure as the current bookmarks menu does (automatic grouping)?	

Continued on next page

ID	*Firefox ID*	**Summary/Description**	**Status**
31		Max number of tabs until arrow buttons appear should be higher. Most of the time I have like 20 tabs open. Right now I have a resolution of 1280x1024. The bad thing is that I can only see 6 tabs at once. If I want to view a tab somewhere between 8 or 20 I have to go through all the tabs (with ctrl-pagdown, or clicking the arrow on the right) to look for the tab I want. It takes a lot of time, while I can just recongnize the tab by it's icon and/or first three letters of the <title>.	
32		do not scale shortcut icons on the bookmark toolbar. the bookmark toolbar should be at least tall enough to display unscaled icons (ie: allow at least 16 pixels for the icons themselves) currently some icons look very bad when they are scaled	

Correlation Graphics

The following figures show the correlations among the characteristics of subjects profile and the dependent variables (time spent to analyze bug reports and amount of duplicates avoided). The meaning of the graphics is that there is no correlation among these characteristics and the dependent variables.

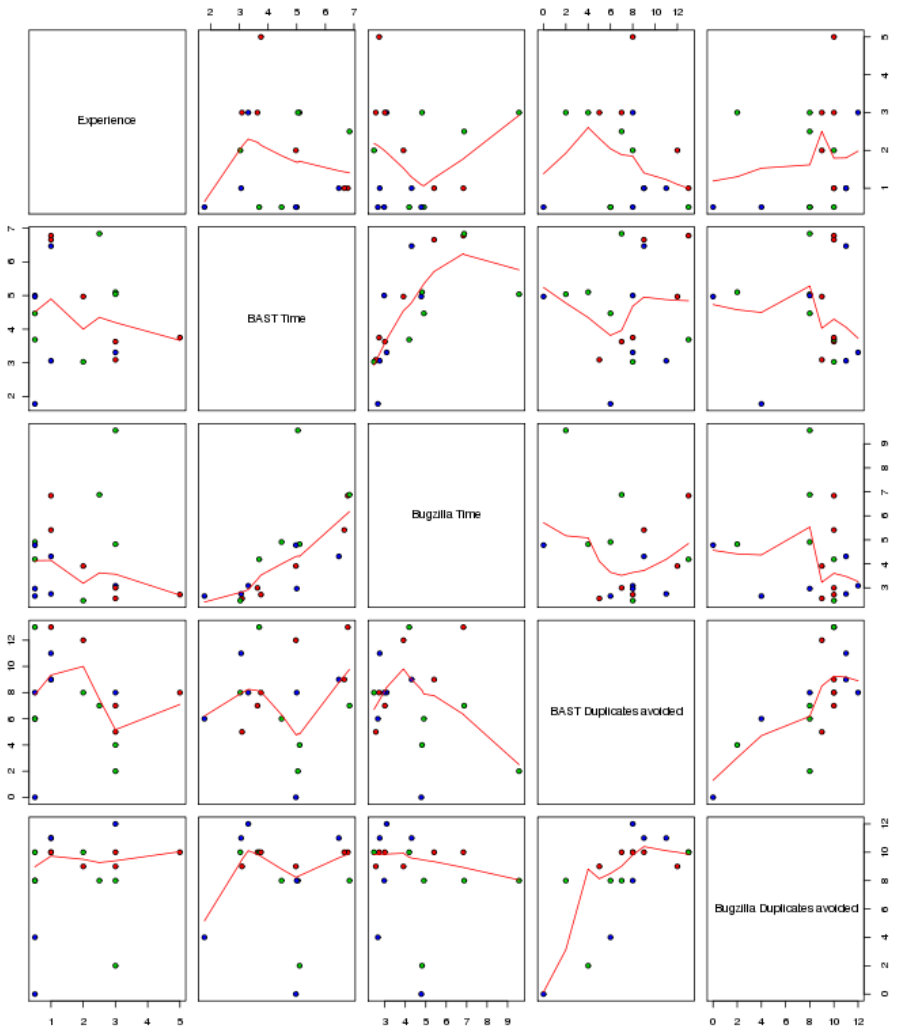


Figure C.1 Correlation among experience and dependent variables.

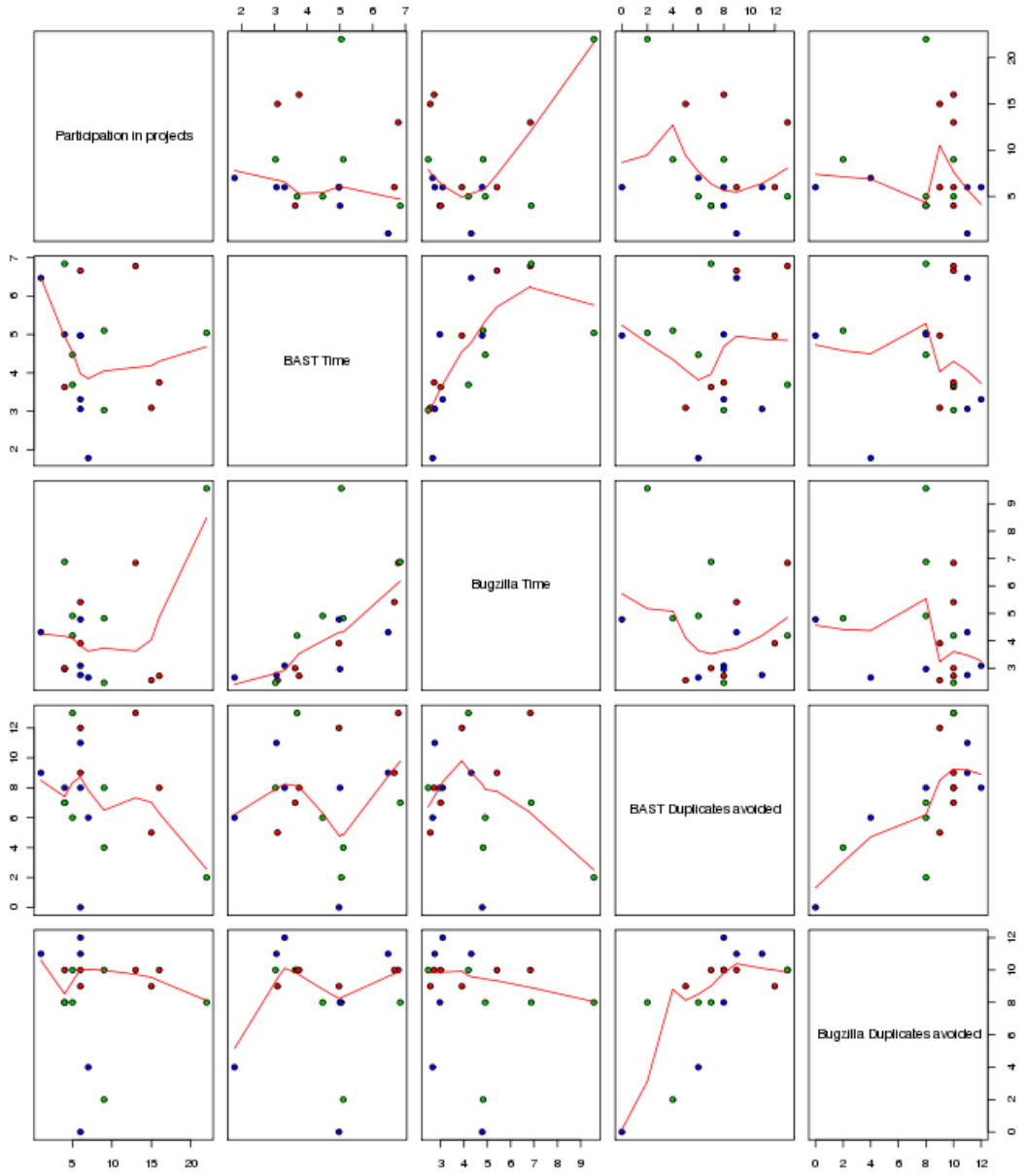


Figure C.2 Correlation among participation in projects and dependent variables.

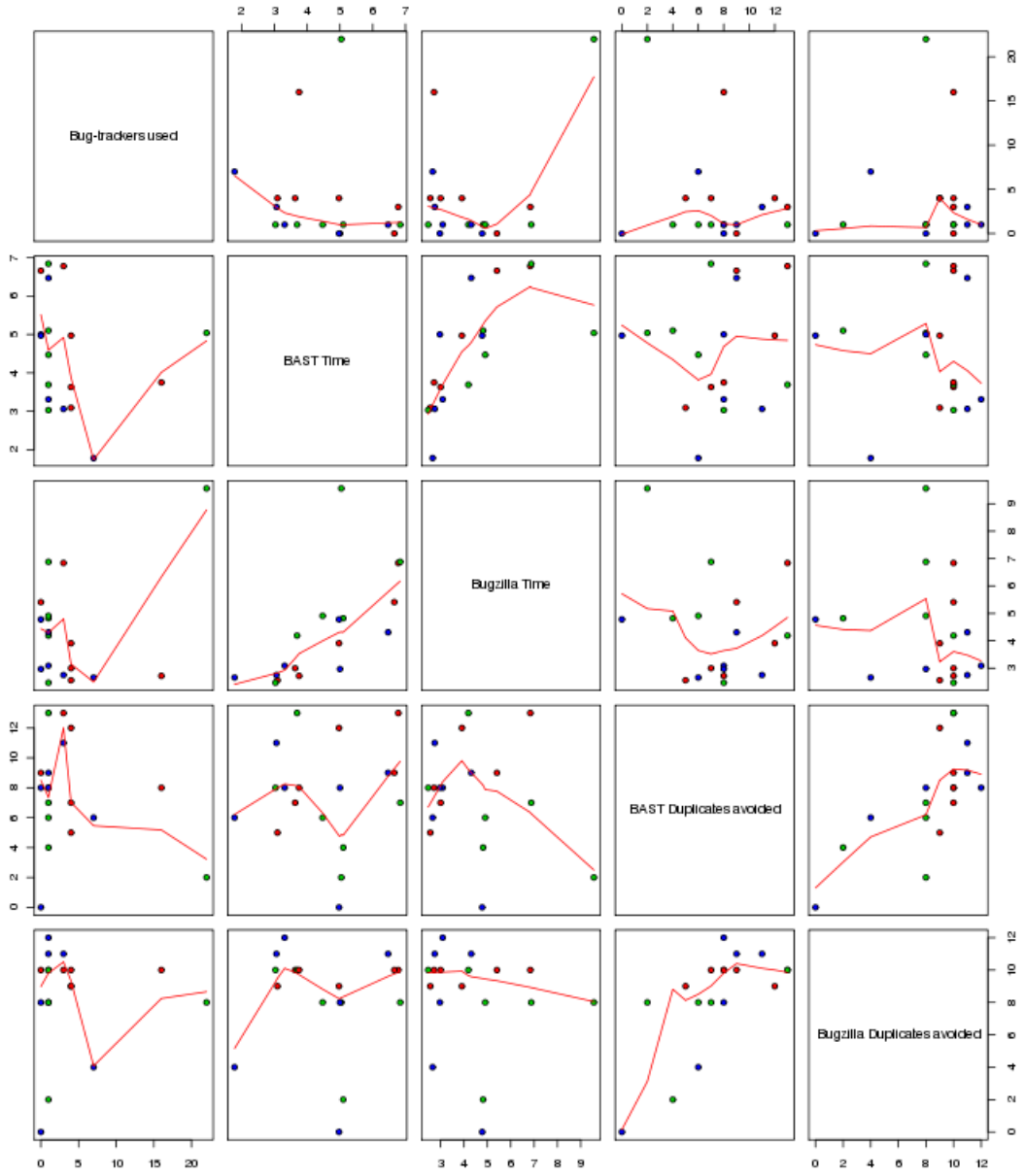


Figure C.3 Correlation among bug-trackers used and dependent variables.