UNIVERSIDAD REY JUAN CARLOS

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Global and Geographically Distributed Work Teams: Understanding the Bug Fixing Process and Potentially Bug-prone Activity Patterns

DOCTORAL THESIS

Daniel Izquierdo Cortázar

Ingeniero en Informática

Madrid, 2012

Thesis submitted to the Departamento de Sistemas Telemáticos y Computación
in partial fulfillment of the requirements for the degree of

Doctor europeus of Philosophy in Computer Science

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, Spain

DOCTORAL THESIS

GLOBAL AND GEOGRAPHICALLY DISTRIBUTED WORK TEAMS: UNDERSTANDING THE BUG FIXING PROCESS AND POTENTIALLY BUG-PRONE ACTIVITY PATTERNS

Author:
Daniel Izquierdo-Cortázar
Ingeniero de Informática - Computer Science Engineer

Director:
Jesús M. González-Barahona
Doctor Ingenierio de Telecomunicación - Doctor Telecommunication Engineer

Co-director:
Gregorio Robles-Martínez
Doctor Ingeniero de Telecomunicación - Doctor Telecommunication Engineer

Fuenlabrada (Madrid), Spain, 2012

DOCTORAL THESIS:	Global and Geographically Distributed Work Teams: Understanding the Bug-fixing Process and Potentially Bug-prone Activity Patterns
AUTHOR:	Daniel Izquierdo-Cortázar
DIRECTOR:	Jesús M. González-Barahona
CO-DIRECTOR:	Gregorio Robles-Martínez

The committee named to evaluate the Thesis above indicated, made up of the following doctors:

PRESIDENT:	Prof. Dr. Baltasar Fernández Manjón (Universidad Complutense, Madrid, Spain)
VOCALS:	Prof. Dr. Natalia Juristo (Universidad Politécnica, Madrid, Spain)
	Prof. Dr. Cornelia Boldyreff (University of East London, London, UK)
	Prof. Dr. Markus Pizka (Itestra Software Productivity, Munich, Germany)
SECRETARY:	Prof. Dr. Luis López Fernández (Universidad Rey Juan Carlos, Madrid, Spain)

has decided to grant the qualification of

Fuenlabrada (Madrid, Spain), April, 2012

The secretary of the committee.

To my family and friends
for their patience and support

Acknowledgements

It is usually difficult to write acknowledgements. Some people decide to write some paragraphs from their heart, while some others do not even write anything at all. I just write without thinking about that. Indeed this part of the dissertation will be probably the most read at all, which is also quite fun after almost six years of work, where I have procrastinated a lot, but worked a bit more.

Probably some people will tell me that I forgot someone important, and I apologize in advance for this, but if you were part of this, you know that I actually know that!. I have met incredible people during this time. First of all my advisors: Jesús and Gregorio. Without them, this dissertation would have been impossible at all. This is of course extensible to all of the people that I have met in LibreSoft like the old guys: Santi, Luis, Andradas, Isra, Teo, Juanjo, Acs or Kal and the not-so-old-guys like Alicia, Gato, Roca, Felipe, Olmedo, Santi (another one!), Pedro or Jorge.

I would like to specially mention the infinite knowledge that I have acquired thanks to the research stays that I have done abroad. Those have allowed me to improve in several ways: language, research, knowledge and culture. I really think that this is probably the best part of the PhD process. Special thanks to Andrea Capiluppi, Cornelia Boldyreff, Tom Mens and Juan C. Fernández Ramil. Their attention and help was invaluable during my time with them.

From a more personal side, I want to thank to my parents, brother and Laura. Their support and understanding has been also invaluable. It is hard to do a PhD, but it is probably more hard to know someone who is doing a PhD and try to have some time with him!.

In any case, like everything in life, there are cycles and this is close to the end. Perhaps the beginning of an academic career, perhaps the beginning of a more industrial one. Who knows and who cares!. At least I finished the PhD!.

Thank you guys ;).

April 2012.

Abstract

The open nature of FLOSS (Free/Libre/Open Source Software) projects has brought new ways of analyzing from different perspectives how software is developed. Indeed FLOSS projects represent a significant portion of the current empirical academic literature.

One of those innovative ways of doing research is related to the mining of software repositories where traces of the activity are kept and publicly available to anyone interested. This also brings positive consequences in contrast to traditional research and this is the possibility to replicate other studies.

With this respect, this dissertation has focused on better understanding the bug life cycle and human related factors that may be more prone to introduce errors in the source code. Specifically, this will play with the concept of *bug seeding* that is the event of causing a later fixing action (*bug fixing commit*). Understanding when a bug was seeded, where exactly in the source code and by whom, opens new research questions that stress the point of looking for potential causes of that involuntary bug seeding event. Although others could have been studied, this dissertation has focused on two potentially buggy attributes of developers that may help them to be less or more prone to introduce seeding commits: experience and the time of the day.

Regarding to the main contributions, from our best known, this is the first study where the concept of bug seeding commits is used to study the bug life cycle: studying the time to fix a bug from its very beginning (when it was *seeded*), resultant distributions, comparison to the time to other similar studies focused on the bug tracking system, finding rate of the buggy commits and study of the defective fixing changes.

Finally, once the bug life cycle was studied, this dissertation looked for potential human related factors that might be more prone to be buggy. Among other results, on the contrary to other similar and previous literature, no relationship between the experience of a developer and the *quality* of the changes was found. Regarding to the time of the day to fix a bug, specific timeframes of the day were found to be more prone to introduce seeding changes than others.

Resumen[1]

La naturaleza abierta de las comunidades de software libre ha permitido a los investigadores analizar desde diferentes perspectivas como se lleva a cabo *in situ* el desarrollo software. De hecho los proyectos de software libre representan hoy en día una buena porción del total de estudios empíricos que se desarrollan a nivel académico en ingeniería del software. La minería de repositorios de desarrollo software es una de esas ramas de carácter innovador que ha aparecido en los últimos años.

Dentro del desarrollo software, esta tesis se ha enfocado en el estudio del ciclo de vida de los errores en el código fuente y factores de tipo humano que puedan ser potencialmente erróneos. A lo largo de la tesis se estudiará el concepto de *bug seeding commit*, momento en el que se introduce un cambio en el código fuente. que posteriormente es detectada como parte de un error que será solventado en una nueva modificación, denominado *bug fixing commit*. Por lo tanto, conocer cuándo una modificación del código fuente provoca el arreglo de un error, dónde y quién fue el autor, ayudar a entender el origen de los errores. A este respecto, se estudiarán dos atributos que dependen de los desarrolladores y que pueden ser sospechosos de introducir nuevos errores: la experiencia y la hora del día en el que se realizan cambios en el código fuente.

Las principales contribuciones de esta tesis en el avance del estado del arte se pueden resumir en dos partes. Por un lado, el estudio de *bug seeding commit* y cómo forma parte del ciclo de vida de los errores, enfocando los estudios en el tiempo medio de arreglo de un error, distribuciones de tiempos resultantes, comparación con otros estudios basados en el sistema de gestión de erratas o estudio de cambios que solventan errores y que introducen nuevos *seeding* commits. Respecto a los atributos de carácter personal de cada uno de los desarrolladores, los resultados no han encontrado relación directa entre la experiencia y una mejora en la proporción de *seeding* commits. Sin embargo, el momento del día cuando se realiza una modificación en el código fuente parece indicar que hay ciertas horas donde esa proporción aumenta, introduciendo más *seeding* commits que la media.

[1] En el apéndice C se puede encontrar un resumen suficiente en castellano que cumple con los requisitos del artículo 24 del capítulo V de la "Normativa para la Admisión del Proyecto de Tesis y Presentación de la Tesis Doctoral" para las tesis que sean presentadas en otros idiomas diferentes del español, como es el caso de ésta.

Contents

List of Figures

List of Tables

List of Acronyms

- **API**: Application Programming Interface
- **BMI**: Backlog Management Index
- **BSR**: Bug Seeding Ratio
- **BTS**: Bug Tracking System
- **CMM**: Capability Maturity Model
- **CMMI**: Capability Maturity Model Integration
- **CPU**: Central Processing Unit
- **CVS**: Concurrent Version System
- **DOM**: Document Object Model
- **DPP**: Defect Prevention Process
- **DQ**: Discussion Questions
- **EA units**: Experience Atoms units
- **FLOSS**: Free Libre Open Source Software
- **FN**: False Negative
- **FP**: False Positive
- **GB**: GigaByte
- **GMT**: Greenwich Mean Time
- **GNOME**: GNU Network Object Model Environment
- **GNU**: GNU's Not Unix!
- **GUI**: Graphical User Interface

- **HTML**: Hyper Text Markup Language
- **ID**: Identifier
- **IDE**: Integrated Development Environment
- **IPC**: Inter-process Communication
- **IRC**: Internet Relay Chat
- **K-S**: Kolmogorov – Smirnov
- **KDE**: word play from Common Desktop Environment
- **OS**: Operating System
- **PDT**: Pacific Daylight Time
- **QA**: Quality Assurance
- **RAM**: Random Access Memory
- **SCM**: Source Code Manager
- **SEI**: Software Engineering Institute
- **SLOC**: Source Lines of Code
- **SQL**: Structured Query Language
- **SRDA**: SourceForge Research Data Archive
- **SVN**: Subversion
- **SZZ Algorithm**: Śliwerski, Zimmerman, Zeller algorithm
- **TN**: False Negative
- **TP**: True Positive
- **UNIX**: Initially named as Unics – Uniplexed Information and Computing Service
- **URL**: Uniform Resource Locator
- **WYSIWYG**: What You See Is What You Get
- **XML**: Extensible Markup Language
- **XPCOM**: Cross Platform Component Object Model
- **XUL**: XML User Interface Language

Chapter 1

Introduction

> ‘‘To be surprised, to wonder, is to begin to understand’’
> ‘‘Sorprenderse, extrañarse, es comenzar a entender’’
> **José Ortega y Gasset** - *La rebelión de las masas*

Consider a long term project with a well defined core group of developers and a set of people contributing seldom during several months of work. A common team of developers tends to work on the source code adding new features, fixing bugs and removing old source code which is not being used in the following. There are also other requirements, such as documentation, artistic facets or translations.

Even those developers with a higher expertise in a given technology and with a good knowledge of the idiosyncrasy of the group will introduce or remove source code with some probability of adding new bugs, what introduces a source of problems for the project.

This new inconsistency may be reported in the future by the same developer, by others or by the users of the product adding a new bug report in some specific infrastructure such as a bug tracking system.

Depending on the current development cycle of the final product, the cost of fixing an issue will increase. The more advanced the development cycle is, the more expensive the final cost will be to fix a bug. It is estimated that fixing an issue when a product has been delivered can be up to one thousand times more expensive than if this would have been done in the requirements phase as can be seen in figure 1.1[1].

With this respect, software maintenance is the study of those changes in the source code after delivering a product. The early detection of issues in the source code is a key factor when managing projects and estimate further errors. It is generally assumed that the maintenance costs will be around a 70% and the development costs around a 30% [Sommerville, 2006].

[1] This is a version of the figure found at [Pressman & Pressman, 2004]

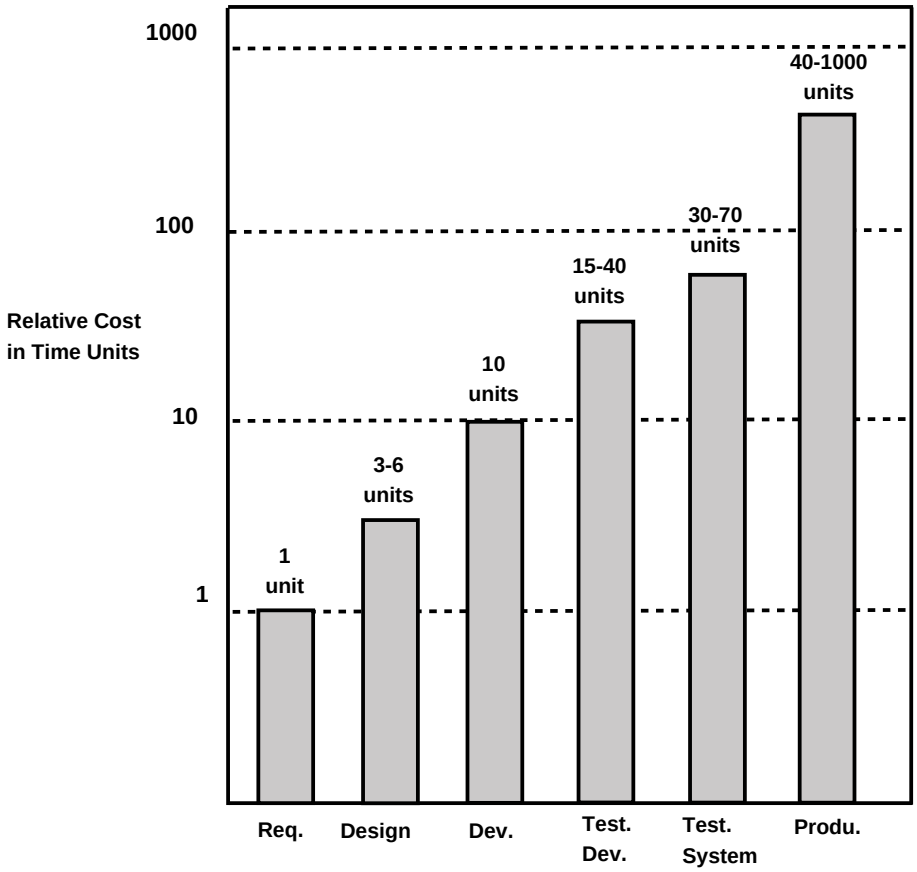

Figure 1.1: *Relative fixing issues cost depending on the stage of the project.*

Those maintenance activities are usually divided into the following subsets [Pressman & Pressman, 2004]:

- **Adaptative**: changes done to adapt the software to a changing environment.
- **Perfective**: changes done to improve performance or maintainability of the product.
- **Corrective**: this type of events correct discovered problems.
- **Preventive**: changes done in the source code to prevent potential issues in the future.

Besides if a developer has fixed an issue, it means that she, another developer or other set of developers have *previously* and *involuntary* introduced that error in the source code. In the following, this action is referred through this dissertation as a *bug seeding* activity, although other types are also plausible such as *buggy* change, *bug injection* or others.

In addition to the research questions depicted later in this dissertation, inside the software quality field there are specific metrics that can be used to measure how good a maintenance task is being carried out. Among others, we can mention those related to the

number of issues or problems opened and closed per month, the mean time to fix an issue, the number of defective fixes and others related to the defect removal effectiveness [Kan, 2002].

In the first case, it is necessary to study how issues are being introduced in the source code. And later check their relationship with the rest of *events* that can be found in the system such as opening and closing a report in the issue tracking system or fixing a bug in the source code management system. In the second case, there are factors that may provoke a higher rate of introduction of errors in the source code and those could be potentially related to human interactions. Indeed, according to [Nagappan *et al.*, 2008], [Conway, 1968], the social metrics[2] or organizational metrics are better indicators to predict software failure. Bringing in context again the aforementioned state that around a 70% of the total cost of the software is due to maintenance activities, this represents an essential field of study to reduce the final costs of a software system. And this could be done by means of the study of the potential reasons why an error is being introduced in the source code.

Figure 1.2 depicts part of those reasons where due to process failures, human related factors such as experience, direct human errors or others may introduce an error. Understanding the causes why an error was introduced or "seeded" in the source code will help in later scenarios. Potential decisions may be made to improve the current process or product quality models, and general maintenance activities that are carried out by the set of developers.

Thus, using the previous introduction as a way to contextualize the thesis, the main goal of this can be summarized in the following sentence:

> This dissertation studies bug seeding activities in the whole bug life cycle and maintenance activities carried out by the developers of a community and the study of those human related factors that are more prone to be *buggy* when developing.

1.1 Context

This section aims to extend the previous introduction to bring in context the main fields of research. Those are directly related to the study of the bug introduction events and organizational structure of the development teams. The followings bullets indicates the structure of this section:

1. **Global distributed development**: subsection 1.1.1 introduces the concept of global distributed development. This is a global fact that companies are outsorcing

[2]These metrics are defined as those derived from the organizational structure of the developers team or related to the developers themselves

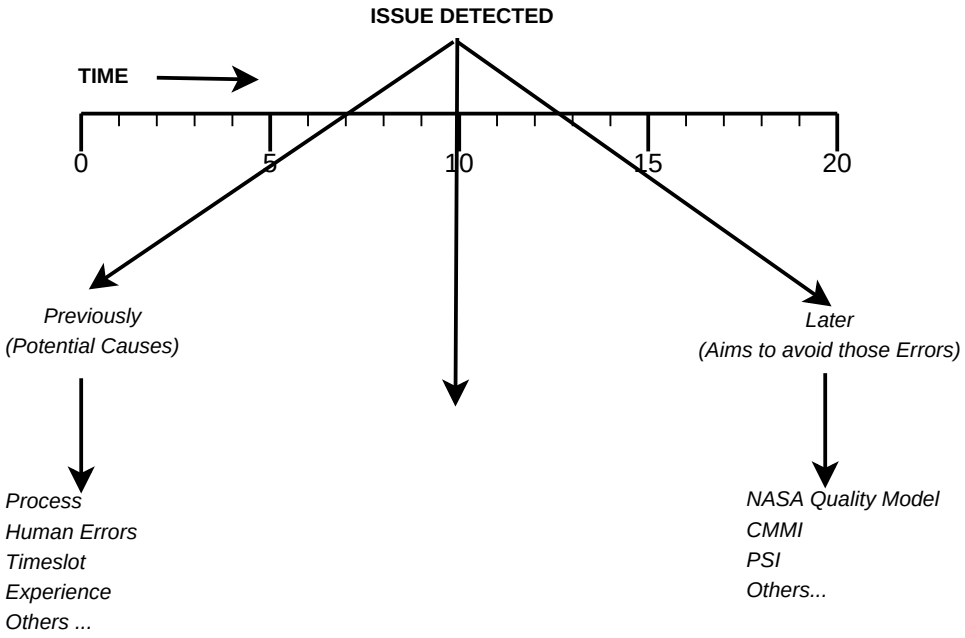

Figure 1.2: *Scenarios affected by the existence of an issue in the source code. Tracing the roots or an issue to study the potential causes of them (Previously). And improving the software processes applying the lessons learned (Later).*

work to other companies and this is even more clear in other communities such as those basing their activities around a common mission such as libre software communities.

2. **Organizational structure**: subsection 1.1.3 will introduce the context of potential human related factors such as the organizational stability, developers turnover or experience and the related problematic.

3. **Software maintenance**: the potential applicability of the thesis is introduced in subsection 1.1.3 where the field of software maintenance is explained and more specifically the bug life cycle and the importance of the early detection of bugs.

4. **Definitions**: along this dissertation, specific vocabulary and concepts will be used. Most of them are introduced in subsection 1.1.4.

1.1.1 Global Distributed Development

Software development demands a highly intellectual work from the the development team: as software evolves there is an increase of its size and complexity, and the underlying source code (among other artifacts) is more likely to get bugs introduced. This is due to the many changes stacked on the source code by the developers of the project. These changes are counted by the hundreds or the thousands, altering the quality of source code,

and statistically increasing the likelihood to inject "buggy" source code into the main system.

With the advent of distributed software engineering, software development has been adapted from the traditional hierarchical approach to a more flat structure, facilitating the coordination of changes and the collaboration of developers around the globe. Nowadays, FLOSS (Free/Libre/Open Source Software) projects are an example of this kind of distributed development[3]. This global distribution of effort is being introduced in enterprises, in a more traditional way: outsourcing [Jiménez *et al.*, 2009] or in a more "FLOSS" way[4] where developers can be found at almost any place around the globe [González-Barahona *et al.*, 2008].

In any case, in both types of communities there exist specific challenges that developers have to deal with [Jiménez *et al.*, 2009], [Herbsleb & Mockus, 2003], [Ebert & Neve, 2001] although some authors have given pieces of advice to deal with this approach and the difficulties that appear related to infrastructure, managing volunteers, licensing or other aspects [Fogel, 2005] and [Bacon, 2009].

In addition, FLOSS communities show an advantage in terms of information provided to the academic community. Their data sources are publicly available what makes easier for researchers to study those projects. This study is focused on this type of communities, but the results and the methodology are also applicable over other kind of communities. Specifically, this dissertation is focused on the source code management system (SCM) where the development effort is centered since this is the place where all the changes in the source code are registered. Besides, partially in the bug tracking system (BTS) where the issue reports are stored.

Although this thesis is focused on two data sources, other tools are used by developers and users. Those can be extended to the basic following:

- Source Code Management System: the best known in libre software world are CVS, Subversion, Git, Mercurial or Bazaar. This type of tool facilitates the interaction when a large set of developers work in the same project, controlling the different versions of the source code. This is even more useful when the developers team is highly distributed around the world and working in different timezones.

- Mailing lists: this provides an asynchronous way of communication among developers and users. These are usually the places where design and architectural decisions are taken. Some others are used as the place where direct communication between

[3] I will assume FLOSS as innovative with this respect along this dissertation since most of the traditional software engineering has been partially modified by new facts coming from FLOSS projects during the last years, although FLOSS projects exist since the 70's.

[4] In projects such as Apache, MySQL, Qt libraries, the Linux kernel and others, where companies play at the same level than other actors such as the FLOSS developers.

developers and users is undertaken.

- Bug Tracking System: also named as Issue Tracking System. This is where errors or patches are reported in order to be fixed or applied by some developers with commits rights. Both, users and developers interact by means of these tools which act as a central point for all of the members of the community.

- Wikis: Collaborative work is done here where specific information about the project is covered. This is useful among developers but also to report new issues to users and other potential interested people in the project.

1.1.2 Organizational Structure

FLOSS communities take advantage of those channels of communication and this facilitates the process of development and communication among members. However any small piece of source code is not trivial and developing demands a highly intellectual work and this is an activity intense in human resources. Besides, the lifespan of a project can range from months to several years or even decades.

Thus, developers typically make changes to the source code while this is being used and evolving. And those changes are potentially buggy changes. In other words, those changes could be named as the seed of a future bug, that perhaps will not be detected in several hours, days or weeks. Or even more, this change is not a bug by itself, but the source code lines added or modified could tend to be (matched with others) a potential source of bugs.

Some authors have addressed the idea of classifying software changes in "clean" or "buggy" looking at the source code [Kim *et al.*, 2008b]. But this type of analysis should be expanded to understand other factors related to the community: developers involved in the process could face a lack of expertise, stress because a deadline is close to the end or some other factors that may affect the final quality of the source code. Those factors can be named as social factors [Nagappan *et al.*, 2008], that are out of the typical metrics related to the prediction of bugs in the source code such as complexity or others.

This idea of how the organization affect the source code has been developed in the traditional literature by Conway [Conway, 1968], Fred Brooks [Brooks, 1995] or Lehman [Lehman, 1979 1980] in different contexts, but based in the same idea: organizations structures and their effects in the overall quality of the source code:

- **Conway's Law**: "...organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations."

- **Lehman's fourth law - Conservation of Organization Stability**: "The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime"

- **Lehman's fifth law: Conservation of Familiarity**: "In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain them."

- **Fred Brooke - The Mythical Man-Month**: As argued by [Nagappan] referencing as a brief summary the classical book by Fred Brooke [Brooke]: "... the product quality is strongly affected by organization structure..."

The three of them based their approach in the quality of the resultant product based on organizational problems. As said in [Nagappan *et al.*, 2008]: *"there has been little or no empirical evidence regarding the relationship/association between organizational structure and direct measures of software quality like failures"*. And in addition, to the best of our knowledge, there has been little or no empirical evidence regarding this concept in FLOSS communities in extension.

One of the problems that are directly related to the Lehman's laws is the regeneration of developers. While old developers tend to leave the project new set of people are attracted and join it having a regeneration of them. It is estimated in at least six months to get some progress and experience in performing at the same rate as an old member of the development team [De Marco & Lister, 1999]. Even more when the development team is working on some more sophisticated work. Unfortunately, experts in specific fields are not really known by their abundance. It means that when some of your experts, as a project manager, leave the project, you may face new problems and a knowledge gap is left.

Although maintaining the current development team could be thought as a plausible solution to mitigate this problem, turnover is usually unavoidable. Being a highly intellectual work, developers have a tendency to lose the original motivation on the software system as time passes by, and they have the natural desire to search for new objectives. In this sense, high turnovers have been observed in most large FLOSS projects, where several *generations* of successive development teams have been identified [Robles, 2006a]. Although these environments are partially, if not mostly, driven by volunteers, turnover in industrial environments is also high.

Expenses in the project probably are the same as above if another expert is hired, however her expertise is not the same, just because we were talking about team, domain or bureaucracy context. Thus, it is almost impossible to find a new developer as skillful as last one and the new one will need some adaptation process.

1.1.3 Software Maintenance

As a summary developers play an essential role in projects since they are the ones implementing new functionalities, maintaining the source code and other similar tasks. They are also the ones who introduce errors in the software even when they are fixing others. Those activities which introduce errors in the source code that were defined as "buggy" activity patterns and can be seen as an extrapolation from the bad code smells which is found to have a huge set of research as seen in the systematic review done by [Zhang *et al.*, 2010]. Bad code smells are those areas in the source code found to be potentially problematic based on previous experience and they are obtained to focus the refactoring effort to avoid future problems. Thus, this concept is extrapolated to the FLOSS communities, where the bad code smells are now the bad activity smells or bug seeding activity and instead of looking for specific patterns such as duplicated code or pointers this study looks for specific community patterns derived from the developers activity, based on frequency and distribution of changes, aging of the developers in the project or developers involved in a set of changes.

Thus, for the software maintenance field, the study of these potential causes of bug introduction activities related to human factors is key to avoid bad practices when developing and carry out policies to avoid further bugs.

If the process of introducing involuntary bugs in the source code is better understood, preventive actions could be taken by project managers or community managers. Thus, it is possible to study the roots of the errors that are discovered by users or developers. It is also possible to improve the ratio of defective fixing changes [Kan, 2002] avoiding or at least peer reviewing those changes that are more prone to be "buggy". And improve in general the performance of the software maintenance metrics or the software processes when developing and working in a distributed environment [Kan, 2002].

1.1.4 Definitions

Finally, before entering in the rest of the dissertation, this study is based on (and could be extended to) projects which use a distributed SCM system called *Mercurial*. For each of the projects analyzed, we have studied the log provided by each of the named SCMs. For this purpose, these are the definitions that were used in the following empirical study:

- **Time of commit**: this is the time when a commit took place. As mentioned above, the present study is based on the *Mercurial* Source Code Management (SCM) system that provides an advantage against other SCMs such as CVS or SVN: the dates saved for each commit are the ones local to the developers. This is a remarkable fact since centralized SCMs (such as SVN and CVS) only store the time when the commit was registered by the server. This facility is quite important in order to directly

obtain the local time when a committer submitted the change to the repository. In all of the selected case studies, the projects started to use the Mercurial SCM from scratch, although a subset migrated to Mercurial from another SCM.

- **Commit (or revision)**: this is a change on the source code (or other artifacts) that is generally done by a developer, although automatic *bots* exists that can also commit changes. A commit stores all the information about the owner of the change, the affected lines and files, and the local date.

- **Bug-fixing commit**: this is a special commit fixing a bug in some specific source code lines, as documented by the committer in the commit log.

- **Bug-seeding commit**: this is a commit (or set of commits) where the same source code lines affected in the bug-fixing commit were either firstly introduced or modified (i.e., *seeded*). From a broader perspective, seeding commits could be seen as those where an issue in the source code has been *introduced* or at least, that those lines are part of the problem, causing a later fixing action.

- **Bug-seeding ratio**: for a given period, this is the ratio between the number of bug-seeding commits and the total number of commits in that same period.

- **Committer**: person who has the right to upload changes to a specific SCM. In the specific case of the Mercurial SCM, there is not a real differentiation between authors and committers, being all of them actual authors (this issue is explained in more detail in the "Threats to Validity" section 5.1). Thus, in this study we will use the term committer, developer or author as synonyms, however, they should be clearly distinguished when using other SCMs.

- **Core committer**: set of committers who usually develop around a 80% [Mockus *et al.*, 2002], [Koch & Schneider, 2002] of the total changes for a given period of time.

- **Line**: at a lower granularity, this is the basic piece of information of this study. Lines are generally handled by committers and in each commit, associated to different files. These lines could present three states: added, modified or removed.

1.2 Research Goals

Considering the organizational structure as a potential source of bugs, several research questions have been raised to be considered in this dissertation. All of them are related to the analysis through the time of the bug-seeding behaviour of the different actors that take place in the development process.

- Study of the event of bug seeding and bug fixing activity in a globally and geographically distributed community of developers. In some works, the bug fixing activity is used as a potential source of prediction for further issues. Or in other cases, the bug related literature is focused on the bug tracking system and not the source code management system. Thus, this research goal aims to study the event of involuntary introducing an error in the source code and how this evolves throught time.

  There is also a clear relationship between the event of introducing a bug, discovering it (opening a bug report), closing it (closing the bug report and changing the correspondent source code to fix it).

- As previously detailed, there exist potential human related factors that may introduce specific errors. Those are divided between the study of the familiarity with the source code using the term experience, and the potential issues derived from the time of the day:

  - "Familiarity" with the source code: in some works, it has been detected a high territoriality of the developers when modifying source code [German, 2004a] and, in addition, there is a natural turnover of the developers [Robles & González-Barahona, 2006] and so on a loss of knowledge [Izquierdo-Cortazar *et al.*, 2009]. However, it is still not well understood the relationship between the developer's experience and the introduction of buggy changes where all of the aforementioned factors are involved.
  - Deviation from typical patterns in project: There are specific timeframes of the day where developers tend to work on the source code. For instance, paid developers will probably tend to work in a more constant way if compared to volunteers (which tend to be the big mass of developers). This common activity patterns could be named as "comfort" timeframes that could be broken when specific issues are faced. Among those, deadlines, changes in the organizational structure, tiredness and others could be mentioned.

1.3 Contributions of the Thesis

The contributions of the thesis can be summarized in the following:

1. Study in depth of the bug introduction process. In addition to the literature where most of the related work is based on studying the datasets provided by the bug tracking systems (normally the time when a bug was open, assigned, tossed, fixed (or not) and closed) and the time when the fixing commits is done; this dissertation

is focused on the time when the error was involuntary introduced in the source code. This event is compared to the rest of the typical events studied in the literature. Specific contributions are the following:

- Comparison of the time to fix a bug in the source code, considering the bug seeding and bug fixing events and in the bug tracking system, considering the open report and close report events.
- Demographical studies of the population of the bug seeding commits. And quick they are being fixed. This provides an estimation on the finding rate depending on the project analyzed. In some cases it has been detected a linear finding rate, while in others this finding rate behaves in an exponential decrease.
- Number of fixing actions that are later discovered to have introduced an error in the source code. Or in other words: study of the defective fixing changes.
- Discussion about the policies carried out by the community when identifying core projects and the overall results found.

2. For this purpose, a new tool was necessary to be created basing its retrieval process in the differences provided by the source code management systems (such as Mercurial or Git) where the lines can be tracked.

3. In addition to the study of the bug life cycle, potential causes related to human factors were selected to be in depth studied in order to check their potential effect when introducing bugs in the source code:

   - Familiarity of the developer with the source code and its relationship when introducing errors. This is measured using several ways of definitions for quantitative analysis of experience: number of commits, number of fixing commits and ownership of the source code.
   - Potential characterization of developers based on their typical activity (mostly focused on maintaining actions or addition of new functionality).
   - Time of the day when the commit is submitted to the repository (taking advantage of the local time provided by the distributed source code management systems such as Mercurial).

1.4 Structure of the Thesis

The structure of the thesis is as follows: this chapter has depicted an overall idea of the main fields of study of the thesis, basically focused on the software maintenance and the

bug life cycle and potential causes of introduction of involuntary "buggy" changes in the source code.

Chapter 2 details the bibliography related to the bug life cycle and organizational metrics phenomenon from a software engineering perspective. First, the literature of the overall retrieval process is detailed. And second, this data mining approach is complemented with specific literature used in each of the research goals: the bug life cycle and the study of potential human related factors that introduce errors.

Chapter 3 is focused on the methodology used along this dissertation which is divided in three main parts. The first one focused on the data mining approach used analyzing the source code management systems (SCMs) and detailing the tools and databases used. Secondly the specific methodology used to fix those commits that are fixing a bug and the later analysis of the time when those issues were involuntary injected in the source code. And finally, the method followed in each experiment is detailed when necessary.

Chapter 4 is the main body of this work and contains the whole set of analysis made using the methodology as a base. In general, these sets of experiments can be divided into three main subgroups. In first place, those related to the study of the phenomenon of *bug seeding* commits which are those that have introduced an error in the source code. Those events are compared to others such as the *bug fixing* commit, the open report action or the close report action (the last two happening in the Bug/Issue Tracking System). The following experiments are related to the study of the potential causes when introducing an error in the source code. Two cases are taken as a case of study: the experience or familiarity of the developers and the time of the day, although other causes are also treated along these experiments.

Chapter 5 specifies the most important results since in the chapter 4 all of the experiments and results are shown. Thus, this chapter is a summary of the whole chapter 4.

Chapter 6 discusses those results with the threats to validity, discussion about the method followed and potential applicability of the results, observations and recommendations done to the Mozilla Foundation and potential further work to be done.

Finally, the appendixes contain extra information related to the validation activities carried out when developing the tool and scripts that have been used and also the replicability of the results together with other minor considerations.

Chapter 2

State of the Art

``[...]we are like dwarves perched on the shoulders of giants[...]''
``[...]nos esse quasi nanos, gigantium humeris insidentes[...]''
Bernardus Carnotensis

Nowadays it is usual to find software companies where part of the work force is distributed into several cities or countries. This global software development has been deployed in different ways depending on the companies. A possibility consists of sending employees to key places. Or in other cases, a company outsources specific parts or the whole development process.

In any case, global software development is raising a new reality related to a global distribution of the effort. Some developers may work from Europe and some other from the USA or China. And the only way to bring developers closer is through the use of channels of communication and the Internet.

FLOSS (Free/Libre/Open Source Software) projects have taken advantage of this distributed environment since the very first years of the very existence of the Internet. In fact, FLOSS projects may be seen as a large subgroup of global and geographically distributed [González-Barahona *et al.*, 2008] developing effort.

However, some challenges are faced by those communities or companies. And most of them related to the communication channels, lack of control and lack of trust which were summarized by [Ramesh *et al.*, 2006] and are listed in the following:

- **Communication Challenges**: in general face to face meetings help to know other members of the team building bridges of confidence.
  - Difficult to initiate communication
  - Misunderstanding/miscommunication

  - Dramatically decrease frequency of communication
  - Increased communication cost - time, money and staff
  - Time difference

- **Lack of control**: from a managerial perspective, the distribution of the effort, even along the 24 hours of the day, difficulties the general management effort.
  - Difficult to control process and quality across distributed teams

- **Lack of trust**: as aforementioned, face to face meetings help to increase the confidence among team members, avoiding undesired situations.
  - Lack of trust between distributed team members
  - Lack of team morale

Focusing on the FLOSS world, teams have partially dealt with this challenges in a satisfying way [Fogel, 2005] [Bacon, 2009]. Lack of trust is solved by the existence of meritocracy in most of the FLOSS projects. And this is the growth professional growth in the community by merits. The better a developer is, the more possibilities she has to get listened. Regarding to the difficult to control process and quality assurance, FLOSS communities have used a peer review process, where most of the important changes are reviewed at least twice.

Regarding to the communication challenges, they are still an issue in most of the communities or enterprises working in a distributed environment. However, defining clear rules about how to participate in the community and in the decisions making process seem to have helped FLOSS communities [Mockus *et al.*, 2002]. Indeed, in order to minimize the impact of the potential challenges regarding to communication channels, FLOSS communities use a set of tools [Fogel, 2005]. Those help to build the community and allow developers and users to interchange ideas and solve problems. This generally creates a network of interest around a project where every member follows the same mission and usually help each other.

On the contrary to the traditional hierarchical structure of companies, FLOSS communities are flatter. The classic *onion model* helps to understand how FLOSS communities are usually formed [Crowston & Howison, 2005]. In first place, a small set of developers are named as *core* developers since they usually carry out most of the work in the community (around a 80% of the total work is done by a 20% of the total amount of detected committers). While there are some other developers who usually tend to be much less productive. The following layers are conformed by end users that in some

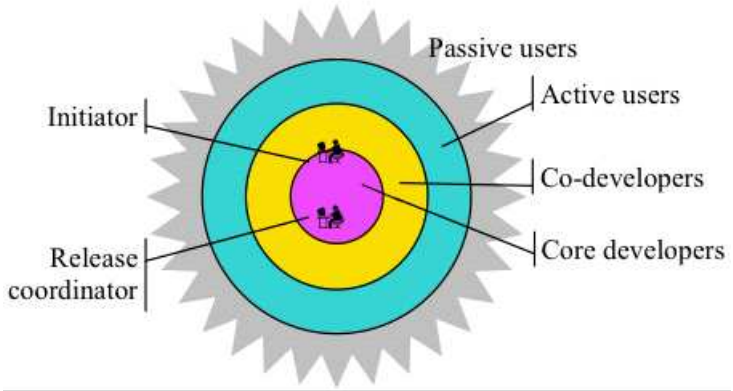

Figure 2.1: *Representation of the FLOSS projects communities: the onion model.*

cases help in the debugging process discovering new issues. Figure 2.1[1] shows an example of this representation.

Coming back to the channels of communication, it is worth to mention that the vast majority of them are publicly available when studying FLOSS projects. And from an academic point of view, they become potential data sources that are available to anyone interested in doing research about FLOSS communities. This process of empirical research is usually named as *mining software repositories* and the method used in this dissertation is detailed in the following chapters.

Finally, regarding to the structure of this chapter: in first place the mining approach is detailed based on previous literature and specific technical reports (2.1.1). Secondly, section 2.2 is focused on the different studies of bugs from different perspectives, mostly in the source code and source code management systems. Finally, section 2.3 focuses on the causes of the "buggy" changes, but from the perspective of human factors that may affect the "quality" of the commits.

In addition, table 2.1 summarizes the most important papers used in the following sections divided by topic.

2.1 Methodology Literature

As a reminder, this dissertation focuses on the study of the bug introduction event. This event is independent from the properties of the source code that was changed. And it is focused on the *organizational* or *social metrics* that are found in the source code management systems (SCM).

Thus, while previous literature was focused on the study of the change itself (e.g.:

[1]Original figure by Crowston and Howison [Crowston & Howison, 2005] that can be found at `http://www.firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1207/1127` - last visited December, 2012.

<table>
<tr><th colspan="2">Methodology</th></tr>
<tr><th>Mining FLOSS data sources</th><th>Detecting seeding-fixing commits</th></tr>
<tr><td>[Robles et al., 2009b],
[Ukkonen, 1985]
[Canfora et al., 2007]
[Chen et al., 2004],
[Śliwerski et al., 2005a],
[Williams & Spacco, 2008],
[Loh & Kim, 2010]
[Canfora et al., 2009]
[D'Ambros et al., 2008]</td><td>[Antoniol et al., 2008],
[Kim et al., 2008c],
[Canfora et al., 2007],
[Kim et al., 2006]</td></tr>
<tr><th colspan="2">Bug Life Cycle</th></tr>
<tr><td colspan="2">[Sommerville, 2006], [Kan, 2002]
[Kagdi et al., 2008], [Sliwerski et al., 2005b]
[Guo et al., 2010], [Capiluppi et al., 2010]
[Hao-Yun Huang & Panchal, 2010], [Pan et al., 2009]
[Gokhale & Mullen, 2010]</td></tr>
<tr><th colspan="2">Human related Factors</th></tr>
<tr><th>Organizational Structure</th><th>Experience</th></tr>
<tr><td>[Crowston & Howison, 2005],
[Robles, 2006b]
[Robles et al., 2009a],
[Nagappan et al., 2008]
[Conway, 1968],
[by Andy Oram & Wilson, 2010]</td><td>[Mockus & Herbsleb, 2002],
[Minto & Murphy, 2007],
[McDonald & Ackerman, 2000],
[Terceiro et al., 2010]
[Izquierdo-Cortazar et al., 2009]</td></tr>
<tr><th>TimeSlot Analysis</th><td></td></tr>
<tr><td>[Śliwerski et al., 2005a],
[Eyolfson et al., 2011]</td><td></td></tr>
</table>

Table 2.1: *State of the art divided by topic*

cyclomatic complexity or programming language) or predicting if a change is more prone to be buggy; **this dissertation knows in advance that a change in the source code is seeding or fixing an issue**. And this will focus on the study of those bug seeding commits and the search of potential causes of introducing involuntary errors in the source code.

This section aims to detail the literature focused on the overall mining activities carried out to retrieve the data necessary to study the bug seeding and fixing activities. This will stress the point on the retrieval information process from the source code management system. And more specifically in the Mercurial repository, used by the Mozilla Foundation, the case of study. With this in mind, section 2.1.1 introduces the FLOSS data mining processes and other literature when analyzing FLOSS data sources. Besides, the general method used has been previously used for similar research and the identification of seeding and fixing commits in the literature is detailed in section 2.1.2.

2.1.1 Mining FLOSS Data Sources

Mining software repositories is an activity focused on the study of several data sources that are used for software development. Among others, several tools have already cited such as the source code management system, mailing lists or bug tracking systems [Fogel, 2005].

A specific case of mining activities is the study of FLOSS projects, where the data sources are publicly available. This opens a large set of possibilities to analyze from different perspectives how software is developed [Robles, 2006b], although this process faces two main challenges [D'Ambros *et al.*, 2008]:

- **Technical challenge**: the existence of several types of data sources increase the complexity and the amount of data to deal with. In addition, the large amount of data accessible by researchers faces scalability problems.

- **Conceptual challenge**: this bullet refers to the way information is treated and later presented to other people interested. Thus, how to present the data to answer specific questions is another challenge.

In order to solve the first of the challenges, there are specific initiatives that aim to integrate the several data sources creating *meta-repositories* of information [Gonzalez-Barahona *et al.*, 2010]. The data is in this case retrieved and pre-treated in order to provide useful pieces of information to researchers.

Among other advantages of this model of using data mined from software repositories, the following one can be cited:

- **There is no need for researchers to retrieve the data**: thus the technical infrastructure needed to retrieve the needed information is not necessary.
- **There is no need for researchers to wait for the data**: as said, the information is already retrieved, so this is directly accessible to everyone at any time. There are specific cases in the retrieval process where to download the data takes days or weeks (e.g.: analyzing a bug tracking system where an user can be banned).
- **Large amount of data is available**: in some cases data from software development activities based on thousands of projects is available.
- **Homogeneous data between researchers**: what facilitates the traceability and replicability of the results.

Among others, the following meta-repositories are some examples[2]

FLOSSMole [Howison *et al.*, 2006], Sourceforge Research Data Archive (SRDA) [Antwerp & Madey, 2008], FLOSSMetrics [Herraiz *et al.*, 2009],

Mining software repositories, thus, is a hard task and meta-repositories of information offer a huge amount of data. However, such as in the case of this dissertation, specific data sources are not offered or have not been previously analyzed, what implies an initial retrieval process of the data. Along this thesis, three tools were used:

- **BlameMe**[3]: stores in a data base the information provided by the *diff* tool run by the source code management system.
- **Bicho**[4]: aims to retrieve information from a given bug tracking system.
- **Cloc**[5]: analyzes the number of source code lines and programming languages.

The last two tools, their use in the dataset is minimum if compared to the use of the BlameMe tool. Besides, the BlameMe tool was the only one that was fully developed in this dissertation and a validation is provided in the annexes of the thesis.

The BlameMe tool retrieves the difference between each pair of revisions focused specifically in the Mercurial repository, and stores those differences in a MySQL database, where the life of each line is tracked from the beginning up to the end. Besides, information from the files, authors, dates and other artifacts are stored.

[2]There exist other initiatives, but not so focused on the main goal of the meta repositories which is to provide software development information to be later used by researchers from hundred or thousands of projects. Among others the NASA metrics data - `http://mdp.ivv.nasa.gov/repository.html` or the Eclipse bug data`http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/`

[3]`http://git.libresoft.es`

[4]`http://git.libresoft.es`

[5]`http://cloc.sourceforge.net/`

With the aim of detailing the retrieval process, the history of the diff tool is introduced in the following subsection. In addition, different papers that have used a similar approach and their purposes will be mentioned together with specific detected limitations.

Brief History of the *diff* Tool

The basic algorithm used in this thesis is based on the *diff tool* that is theoretically explained by [Ukkonen, 1985], [Miller & Myers, 1985] and [Myers, 1986]. This tool is fed by two different version of a file or by two different version of the same directory and returns the differences found between them. The *diff* looks for *plain* differences and this provides information for both, the *real* differences and a way to ignore *apparent* differences such as spaces or indentations.

A known implementation of this algorithm is the GNU, detailed at [MacKenzie *et al.*, 2002] However, each source code management system usually implements their own diff tool or re-use the GNU version. For instance, the Git SCM has implemented one, and the Mercurial community has developed another one. All in all, all of the SCM use the *unified* format as the output of their diff tool and this simplifies the approach and whose results are the same as using the GNU diff tool.

Purposes of studying at the line level

The main question now is: why is this useful?. In most of the cases, the diff tool provides extra information about a specific commit. However, researchers have worked with the granularity of commit or a file. And there are few works using the granularity of a line.

As specified by [Canfora *et al.*, 2007], these studies could be used for different purposes:

- **Software evolution studies**: there are plenty of studies based on the software evolution of several systems, however, with this granularity, it is possible to go a step ahead, knowing the real authorship of those lines, what could provide extra information in the terms of *knowledge* of the source code.

- **Effort estimation**: given a specific commit, it is easy to know, how many lines have been added, removed or modified by the author of that commit. In terms of the effort estimation, this provides extra information and could be used, for instance, to create effort estimation models in FLOSS ecosystems.

- **Crosscutting concerns evolution**: this item is related to the idea of how much has evolved a piece of source code. This is also interesting from the patch application point of view, checking how the patches deal with that piece of source code. In any case, a concern is a general idea, so almost any field of the software evolution field could be taken into account and improved with this approach.

- **Clones evolution**: this type of studies also provide specific information of the lines that were copied or moved from one file to another one. This would help to estimate the commit when the clone was undertaken.

In addition, other pieces of information have been identified that could help in the aforementioned purposes of study. Since, the Mozilla Foundation is using a distributed SCM, other attributes are also studied and can help:

- **Real Authorship**: the use of distributed SCMs provides extra pieces of information that can be mined and later studied. This is the case of the differentiation made between author and committer in Git repositories. Or the direct use of author instead of committer in the case of Mercurial repositories. However, older SCMs such as CVS or SVN do not offer this type of information. In this case, developers interested in participating in the community need to create a *patch* with the changes. Those changes will be later uploaded to the server.

  As mentioned, communities tend to follow the *onion* model and the total work is not only done by the *core* developers, but also by people around the communities [Crowston & Howison, 2005]. The authorship of those sporadic changes to the source code are, in the case of distributed SCMs, registered. In other cases, like in CVS or SVN, this information, if provided, is found in the log message left by the committer or in the copyright or AUTHORS files.

  This type of information is quite useful when estimating real efforts since the effect of *gatekeeper* is avoided. Being a *gatekeeper* those developers that commit changes to the source code in behalf of others that do not have commit rights.

- **Time of commit**: the use of distributed SCMs help to easily create branches and clones of the source code. Besides the registered real authorship of the changes, the time of commit (and time zone) is also registered at the moment of submitting changes to the SCM. On the contrary, centralized SCMs only keep the time of commit registered in the server. Thus, extra information related to the timezone of the developer or the actual local time of commit is totally lost.

  This type of information is quite useful when estimating time of work of the developers and type of effort that usually carried out FLOSS communities. As an example, it is possible to determine if the effort is usually undertaken during morning, afternoon or evening timeframes.

Bias in the retrieving process

Data mining activities have been detected to have specific limitations that are directly related to the data sources used in the retrieval process.

This dissertation is mostly focused on the source code management system, although other data sources have been used. This is the case of releases of source code and bug tracking systems. However the only direct match between two different data sources has been made between the SCM and the BTS when linking bug reports to the traces left by developers in the commit log[6].

As said in [Chen *et al.*, 2004], "*...ChangeLog files (and other open source artifacts) were not created for research purposes, but rather to support software development by easing tasks like maintenance and testing'...*". Thus, researchers should be extremely careful when using these data sources.

With respect to the linkage of repositories, it has been studied by several authors that specific limitations should be taken into account when linking these data sources [Antoniol *et al.*, 2008], [Bird *et al.*, 2009a], [Nguyen *et al.*, 2010], [Bachmann *et al.*, 2010]. For instance, lack of information about the issue that was fixed in the SCM or the other way around: lack of information in the BTS about the part of the source code that has been fixed. However the same authors have raised the point that even when the biases exist, this is inherent to the software process and the bug predict models can produce similar results.

In addition, the same authors have not proved yet that a more mature project may better face this bias [Bird *et al.*, 2009a]. Thus, the matureness of a project is not a proved indicator to take into account when biasing the studies. Or in other words, the matureness is a project, does not imply a better relationship across different data sources. Thus, the authors encourage to further studies about taking care when using this type of datasets. Since this dissertation is mostly basing its analysis in the source code management system and not other data source, they only case when this should be taken into account is when matching the bugs found in the SCM to the BTS.

Regarding to the extraction of *diff* data itself, there are other improved methods, such as the one proposed by Canfora [Canfora *et al.*, 2009]. However, there are several implications that forced us to avoid the integration of this improved method in the current toolset and discusses in further work section (chapter 6, section 6.2).

Finally, generalizing the empirical approach used in this dissertation, there is specific literature related to how to do empirical research based on a case study that should be mentioned: [Runeson & Höst, 2009], [Kitchenham *et al.*, 2002] and [Fernandez-Ramil *et al.*, 2008].

[6]The rest of potential threats to validity and limitations are specified in chapter 6, section 5.1

2.1.2 Detecting Seeding and Fixing Commits

A bug fixing commit, by definition, is the change to the source code that has resolved a previous detected issue. In addition, a bug seeding commit is the change to the source code that "has caused that fixing action" [Śliwerski *et al.*, 2005a].

Using the information provided by the diff tool found in different SCM or the GNU diff, seeding commits are calculated by means of the source code lines that were modified or removed in commits detected as fixing an issue. The lines *modified* or *removed* in a fixing commit are the suspicious of being part of the problem. Thus, the life of those lines is calculated and the previous change were those were *added* or *modified* is defined as the point in time where the error was *seeded* (along this dissertation named as bug seeding commit). This algorithm, named as *SZZ algorithm*, was initially proposed at [Śliwerski *et al.*, 2005a] and later improved at [Williams & Spacco, 2008] by different authors.

The method of following the life of the lines has been also used by other authors. Indeed, [Loh & Kim, 2010], [Antoniol *et al.*, 2008] and [Canfora *et al.*, 2007] have studied the behavior of the *diff* command line when tracking differences between each pair of revisions or each pair of files. It is stated that the diff tool works pretty well for the purpose it was designed for. However, in some specific cases, the detection of the lines that were added, removed or modified, is not perfectly done focusing the results on the chunks obtained from that tool[7].

In any case, this method is being used in the literature as a way of tracking the moment (commit) where a bug was introduced. An example is the detection of detection of *buggy* changes in the source code: [Kim *et al.*, 2006] and later improved at [Kim *et al.*, 2008c].

Regarding to these papers, the method proposed by [Kim *et al.*, 2006] aims to predict "*latent software bugs*" extracting information from only the source code management system. Obtaining a 77% of accuracy and a 57% of buggy change recall in the case of the Mozilla community when detecting *buggy* change in the source code.

With respect to the bug fixing activity detection, the authors used the same approach as used in this dissertation: a commit is fixing an issue if this is identified as using the pattern: key word "bug" or "Bug" plus an id reference number.

In addition, when retrieving lines that introduced an error in the source code, the SZZ algorithm is used as done in this dissertation.

Finally, according to the authors, the dataset used for the Mozilla community is focused between August, 2003 and August, 2004, measuring between 500 and 1,000 revisions and detecting around a 30% of buggy changes.

[7] As seen, the Mercurial repository, SCM used by the case study of this dissertation, does not mark those renamed files, as for instance, Git does. However, for performance issues, it was decided to avoid the use of that *ldiff* [Canfora *et al.*, 2007] and extend the analysis to more projects and types of files (as discussed in chapter 6, section 6.2) .

2.2 Bug Life Cycle

This section aims to detail the literature focused on the study of the bug life cycle from the source code management and bug tracking system perspective. According to classic literature [Sommerville, 2006], the involuntary introduction of bugs in the source code is an unavoidable task. It is inherent to the process of making software. With this respect, process and product quality models have been defined, aiming to improve the general software development process and reduce the number of bugs in the source code.

FLOSS projects usually leave the management of issues to specific tools such as the Bug Tracking System (where one of the best known is the Bugzilla tool [8]). This type of infrastructure provides a bug life cycle, as seen for instance in the bug life cycle from the Bugzilla project[9], there are several potential resolutions for a given issue:

- **Fixed**: the issue was fixed.
- **Duplicated**: the issue has been detected to be duplicated.
- **Wontfix**: the issue will not be fixed.
- **Worksforme**: the developer could not reproduce the issue.
- **Invalid**: it is not a bug.
- **Remind and later**: these field are not part of the new versions of the Bugzilla tool.

Regarding to the opening steps, in first place this has to be detected by a developer or user and later a bug report will be opened in the specific BTS.

As specified, this dissertation aims to study the bug life cycle adding a new event to this bug life cycle: the time when a bug is involuntary introduced in the source code. Thus, this event is not registered in the BTS, but will be noticed after a while by the developer or user that later will open a report in the BTS.

Thus, the better the bug life cycle is understood, the better is the quality assurance process. And this can be measured by metrics. With this respect, there are three main fields where software metrics can be classified [Kan, 2002]:

1. **Product metrics**: this set of metrics aims to describe the characteristics of a product focusing on the source code lines (complexity, lines of code or coupling among others) or the costumer satisfaction (mean time to failure, defect density or costumer problems among others).

[8] `www.bugzilla.org/`
[9] `http://www.bugzilla.org/docs/2.20/html/lifecycle.html#lifecycle-image`

2. **Process metrics**: this set of metrics aims to improve software development process and maintenance activities. Among others, it can be found the study of causes of bug introduction, effectiveness of defect removal or others.

3. **Project metrics**: this set of metrics aims to describe the project characteristics. As an example, number of developers, cost estimation, software life cycle or others.

The potential applicability of this dissertation is mostly focused on the second set of metrics, where the main goal is to study and describe specific metrics (in this case study the bug life cycle and potential causes of their injection in the source code) and later (as potential applicability) improve the quality process models to avoid specific situations where developers are more prone to be *buggy*.

With this respect, it is worth to mention the Defect Prevention Process (DPP) [Kan, 2002]. This is a process to continually improve the development process and it is based in three main legs:

1. Analyze defects or errors *to trace the root causes*.

2. Suggest *preventive actions* to eliminate the defect root causes.

3. *Implement* the preventive actions.

2.2.1 Time to Fix a Bug

In general terms, the software maintenance field has dealt with the idea of adding the time to fix a bug as a metric when creating models [Kan, 2002]. However, not many of them have deepened in the field of modeling the distribution of time to fix a bug and other aspects.

Gokhale [Gokhale & Mullen, 2010] has pointed this limitation (specifically focusing on the field of software reliability) and has aimed to analyzed in depth this distributions. The time to fix a bug definition used by the author is the "*duration of the time from when the defect is written until the defect is repaired*". The specific time when the bug is *written* happens when the time is discovered and a bug report is opened.

Gokhale defines three main states for the bug life cycle for her purposes: one transitory state and two terminal states:

1. **Transitory states**: the issue is in the first stages of its cycle in the bug tracking system. This still needs to be queued, assigned and analyzed.

2. **Terminal states**: bugs have reached the stage of duplicated, can not be reproduced and other final stages but not the fixed one.

3. **Terminal states**: the bug have reached the stage of fixed or resolved and the patch has been applied in the source code.

The authors claim that, previously, the time to fix a bug was expected to follow an exponential or log-normal distribution. However the results, based on nine products from CISCO Systems and with a size of more than 1 million of source code lines show that the repair times can be characterized by the Laplace Transform of the log-normal distribution. The sample contains a population of 10,000 bugs.

Other authors have studied this metric to build prediction models [Weiss *et al.*, 2007], [Giger *et al.*, 2010], [Panjer, 2007]. In order to build this predictor, the authors deal with the very first information available in the bug tracking systems.

However, all of the previous literature is based on the information available in the bug tracking system. This dissertation aims to also add another variable to the study: the event when a bug is *seeded* in the source code. This opens a new set of questions related to the time to fix a bug. With this in mind, two challenges are faced: studies focusing on the time to fix a bug (basic statistical approach, study of the distributions or others) and comparison of these new results with the current literature (focused on the BTS, and not in the SCM).

2.2.2 Bug Fixing Process

In terms of relating the bug fixing process and its responsibilities, some authors have dealt with the idea of who should fix certain bug [Kagdi *et al.*, 2008], [Sliwerski *et al.*, 2005b], based on previous changes of the same file, or at least slices of the changes introduced in a file. Another approach used to deal with the same problem, has been adopted at the level of the bug tracking system. In a study based on the development of *Microsoft Windows Vista* and *Windows 7*, it has been found that the number of reports *opened* by one developer and initially *assigned* to her development team tend to be fixed more quickly than bugs that are assigned to another development team [Guo *et al.*, 2010]. Finally, it has also been reported that specific FLOSS communities try and reinforce a per-contributor sense of responsibility: in highly modular projects (as for instance *Moodle* or *Drupal*), for example, it is a shared expectation within the community that the original contributor will support his/her modules [Capiluppi *et al.*, 2010] and keep them in sync with the evolution of the core system [Hao-Yun Huang & Panchal, 2010]. Finally, other authors, have dealt with the idea of looking for bug fixing patterns in the source code [Pan *et al.*, 2009] analyzing the different revisions provided by a given SCM system, but focusing on the semantics of the source code. In other words, they are aware of several common fix patterns such as "*addition of precondition check*" or "*different method call to a class instance*".

2.2.3 Studies on the Estimation of Errors in the Source Code

This experiment is directly related in first place, to the bug life cycle, studied from this perspective in this dissertation to better understand the life of a bug. But also from the software maintenance field. In fact, traditional studies estimate the remaining faults in the systems and use this estimation to predict new bugs in the future. However, those models and estimations [Eick *et al.*, 1992], [Christenson & Huang, 1996] do not take into account that "*new faults are continuously being added to the system as changes are made*" [Graves *et al.*, 2000].

This dissertation follows up this type of experiments, where it is accepted as natural the evolution of the software. On top of those experiments, other authors have worked on investigating the natural causes of the errors, trying to reach the *roots* of those errors [Basili & Perricone, 1984], [Hatton, 1997], [Yu *et al.*, 1988] to later improve the software quality assurance and software maintenance process.

And finally, other authors have aimed to go a step ahead and study the organizational or social metrics that may help to improve the estimation models. Those can be listed as those metrics totally independent from the source code and focused on the development teams and their interactions with the source code [Nagappan *et al.*, 2006], [Nagappan & Ball, 2005], [Nagappan *et al.*, 2008], [Kim *et al.*, 2008b].

According to the last set of papers, it seems that the organizational metrics show a better accuracy when predicting bugs in the source code than the traditional ones. Besides, those are based on the study of the history or traces left by developers and later used to predict bugs [Zimmermann *et al.*, 2008].

Finally, as an addition, a subset of those changes to the source code that are *buggy* are the fixes in the source code that later became *buggy* or in other words: defective fixing changes [Kan, 2002]. This has been defined as: "*The metric of percent defective fixes is simply the percentage of all of fixes in a time interval that are defective*".

2.3 Human related Factors

In general terms, development is an activity that requires a high level of specialization from the workers. As any activity that comes from the human side, this is prone to be *buggy* and the software engineering field aims to study and decrease as many as possible the erroneous activity from the developers.

FLOSS communities, as case study and distributed community is a specific case where new variables should be taken into account.

In first place, FLOSS communities follow the well known *onion model* [Crowston & Howison, 2005] where there exist concentric layers divided by the degree of activity and

compromise with the community. Figure 2.1[10] shows this layers schema.

At the center of the *onion* the most active developers are found, reaching around a 80% of the total activity (and being around a 20% of the total number of committers). In the following layers the activity tend to decrease while the number of developers tend to increase. Last layers are occupied by active users that usually provide feedback to the community and finally the rest of the users that simply "use" the software.

In addition, that onion model is dynamic and developers tend to leave the community or the project, while others tend to be in charge of new responsibilities inside them [Robles & González-Barahona, 2006], [Robles *et al.*, 2009a]. Thus, main issues may appear when developers leave a project and specifically, there may be a knowledge gap that may affect the quality of the source code [Izquierdo-Cortazar *et al.*, 2009].

This dissertation is focused on two main concepts to study the human factors, although others may exist. In first place, the experience of the authors when changing the software and in second place the potential problem of working in a continuous 24 hours framework around the globe. These set of metrics could be seen as *social* metrics and those are related to the aforementioned project metrics [Kan, 2002] where other variables are taken into account, more than the traditional product and process metrics.

2.3.1 Organizational Structure and *Social* Metrics

The impact of social metrics, based on the organizational structure [Nagappan *et al.*, 2008] derived from the Conways law [Conway, 1968], seems to be better when predicting potential bugs in the source code if compared to other methods as explained in [Nagappan *et al.*, 2008]: code churns [Graves *et al.*, 2000], code complexity [Khoshgoftaar *et al.*, 1996], code dependencies [Podgurski & Clarke, 1990], code coverage [Hutchins *et al.*, 1994], combination of metrics [Denaro & Pezze, 2002] or pre-releases bugs [Biyani & Santhanam, 1998].

Specifically in the study of the organizational structure of the projects, Nagappan [Nagappan *et al.*, 2008] and Bird [by Andy Oram & Wilson, 2010] have introduced the concept of social metrics as potential failures predictors.

Among others, the following metrics were defined in the project metrics context:

- **Number of engineers**: this is the number of unique engineers touching the source code and currently employed.
- **Number of ex-engineers**: this is the number of unique engineers that have touched the source code and currently not employed by the company in charge of the development

[10] Figure based on the Crowston paper [Crowston & Howison, 2005] and obtained from Robles dissertation [Robles, 2006b]

- **Edit frequency**: this is the total number of modifications in the source code before a release of the source code.
- **Depth of master ownership**: this is level of ownership of the release based on the changes in the source code.
- **Percentage of Org. contributing to development**: this is the ratio of the "Depth of master ownership" metric out of the total organizational size of the project.
- **Level of organizational code ownership**: percentage of changes to the source code from the organization that contains the ownership of the release.

All of the previous metrics were identified as being independent among them, using a statistical approach and based on the Windows Vista development process.The results showed that using this "social" metrics, the percentages when predicting bugs in the source code increased if compared to other methods.

Thus, there are three main variables in these studies: people, ownership and edit frequency. And the authors stated that the study should be extended to FLOSS projects to enrich the data set and study other environments than proprietary systems.

2.3.2 Familiarity with the Source Code

Experience or expertise is a recurrent piece of study in the current literature. As stated by Mockus [Mockus & Herbsleb, 2002]: "*Finding relevant expertise is a critical need in collaborative software engineering, particularly in geographically distributed developments*". And previous work suggests that the first criterion when selecting people with a high expertise is the experience [Ackerman & Halverson, 1998].

However, what does exactly mean experience?: some authors have dealt with the idea of commits done in the SCM [Mockus & Herbsleb, 2002], [Minto & Murphy, 2007] [McDonald & Ackerman, 2000], [Girba *et al.*, 2005]. Others in the recent activity plus other factors such as ownership [Fritz *et al.*, 2007a], [German, 2004a]. And others measure expertise as mostly fixing activity [Ahsan *et al.*, 2010].

In [Mockus & Herbsleb, 2002], the expertise is measured by means of the Experience Atoms (EA units). The experience, thus, is the result of the activity of a developer when changing the source code. Thus, the basics of EAs are the atomic changes to the source code: *commits*. In addition, several contexts were defined to measure software experience:

1. Releases of software, patches, bug fixing activities or others.
2. Functionality of the final product: databases, GUI, or other types of functionality.

3. Infrastructure used to develop the work: IDE's, programming languages or others.

4. Type of change: corrective, adaptative or perfective among others.

In the context of this dissertation, the aim of this study is to quantify experience. Thus, the first of the bullets is the one where all of the information can be found at the SCM. The rest of them can be partially measured using the extra information found in the SCM. For instance, in the case of the programming languages, this is retrieved by means of the extensions of the files touched by a developer.

In [Ahsan *et al.*, 2010], the expertise is defined as a mix of three metrics that help to create the named "log-book":

1. Sum of bug fixed by a developer

2. Weighted sum of bug fixed by a developer. The weighting factor is the severity of the fixed bug.

3. Sum of source code files fixed by a developer.

Thus, there are several ways of defining expertise, but in most of the cases there is a common denominator: the more changes done in the source code, the higher the expertise of a developer. And this is one the type of experience that will be used along this dissertation: *number of commits done on source code files.*

Using this definition of expertise, some authors have studied the correlation between the percentage of *buggy* changes in the source code with the total number of commits [Eyolfson *et al.*, 2011]. It has been observed a correlation between the two variables where the higher the experience measured in number of commits, the less likely the introduction of buggy changes in the source code. However, the authors claim that further studies must be done within this respect.

Finally, regarding the definition of expertise, this is a critical need in collaborative software engineering, even more with the existence of geographically distributed teams [Mockus & Herbsleb, 2002]. This is also interesting when determining the potential developer that best fits to help in a piece of source code [Kagdi *et al.*, 2008]. In this case, the authors use the assumption that those developers who have substantially contributed to specific files in the past, are more likely to better help in new changes to that region of the source code. Indeed, other authors have dealt with the idea of measuring "knowledge" focusing on a qualitative study of the developers [Fritz *et al.*, 2007b]. From this point of view, the results show that the frequency and age of the changes in a set of files can indicate knowledge over the source code. Also, the authors add specific ideas to improve the model such as authorship over the source code among others.

In terms of ownership or authorship, other concepts are brought in context. For instance, the concept of *territoriality* is based on this concept of ownership. Territoriality is measured as the number of files that are touched by just one developer. This is also applicable to other artifacts in the source code such as methods, classes or modules [German, 2004a], [Robles *et al.*, 2006], [Robles, 2006b].

Zooming out, these concepts of experience and territoriality can also be seen as familiarity with the source code. With this terminology, some authors have used the fixing activity as a way to measure familiarity with the source code [Nguyen *et al.*, 2010] and also the well known definition of core and non-core developers given by the *onion* model. Thus, as expected, the more experience, the fewer number of defective changes to the source code. As studied in [Terceiro *et al.*, 2010], non-core developers are more prone to introduce changes in the source code with a higher complexity. And the complexity in the source code is used as a maintenance metric [Kan, 2002].

Finally, regarding to the general concept of familiarity with the source code, this idea is closely related to the Lehman's Law, specifically to the fourth evolution law: "*Conservation of Organization Stability*" and fifth: "*Conservation of Familiarity*". These, have addressed the familiarity and organizational stability as a key point.

Besides, as it is known from a manager perspective, there is a natural regeneration of developers [Robles & González-Barahona, 2006] where some of them tend to leave the project, while others add their effort. In the more specific case of FLOSS projects, those are mostly volunteers and this implies that it is more difficult for FLOSS projects to find an adequate role for each vacancy in a quick way.

Thus, developer turnover can result in a major problem when developing software. When senior developers abandon a software project, they leave a knowledge gap that has to be managed [Izquierdo-Cortazar *et al.*, 2009]. In addition, new (junior) developers require some time in order to achieve the desired level of productivity.

Summarizing, this dissertation will use, as a quantitative way of measuring experience, the number of source code commits, number of fixing source code commits and territoriality.

2.3.3 Software Maintenance Metrics related to Expertise

Closely related to the concept of familiarity to the source code is the seventh law by Lehman: there is a declining quality in the source code. Since the growth of the source is an unavoidable effect of the evolution of the software, it is claimed by Lehman that this continuous increase will cause a potential decay in the quality of the software.

The maintenance effort, thus, is needed together with re-engineering activities. Among others, as specified by [Kan, 2002], several metrics were defined as important when estimating the maintenance effort:

1. **Fix backlog and backlog management index**: this metric is related to the ratio of fixed bugs against the number of new bugs coming to the project.

2. **Fix response time and fix responsiveness**: this is the time that usually takes to fix a bug since this has been discovered.

3. **Percent *delinquent* fixes**: the *delinquent* fixes are defined as those errors that have exceeded their typical time to fix. This time is based on their severity.

4. **Fix quality**: this metric aims to measure how good are the changes to the source code that are fixing an error. This is measure through the percentage of them that are introducing new errors in the source code. Or in other words: the percentage of *defective fixing changes*.

2.3.4 TimeSlot Analysis

The final aspect to take into account in this dissertation when using human related factors is the time of the day. With this respect, the closest works done in terms of time of the day analysis are [Śliwerski *et al.*, 2005a] where the authors declare that Fridays are not good days for Mozilla developers. The highest ratio of introduction of commits causing fixing actions happened during that specific week day.

In addition, other authors have partially developed this idea from the point of view of the time of the day [Eyolfson *et al.*, 2011], focusing on the PostgreSQL and Linux Kernel projects. Their main findings can be summarized in the following:

1. The authors studied the correlation between commit *correctness* with the time of the day, day of the week, developer experience and developers commit frequency.

2. Late-night commits (00:00 - 04:00) present a highest bug seeding ratio.

3. Morning commits (07:00 - 12:00) present the lowest bug seeding ratio.

4. Developers with a higher activity in the community tend to introduce less "buggy" changes in the source code.

On top of the previous findings, this dissertation will try to confirm or do not them, using a similar approach. In order to advance the state of the art, this dissertation will study the resultant general effort aggregation, usual timeframe of activity of developers and specific times of the day where developers are more prone to be affected by bugs.

Chapter 3

Methodology

‘‘If anyone should think of scientific method as a way which leads to success in science, he will be disappointed. There is no royal road to success’’

Karl Popper

The methodology chapter aims to detail several methods and tools used through this dissertation. As previously detailed, the general goal of this thesis consists of understanding the bug life cycle and potential human factors that may be more prone to introduce errors in the source code.

For this purpose, only the source code management system is studied and in some cases the bug tracking system is added to complement the experiments. This helps to limit the range of potential threats to validity when crossing different data sources, but also helps to deepen in the source code management system and understand who, when and potentially why an error was introduced.

As a general overview, the empirical study is based on the study of the commits that have fixed an error. And using the source code management system, it is possible to trace the history of that commit to reach the previous steps that caused that fixing action. That (or those) commit(s) helps to understand when that commit that caused the fixing action was introduced. And this allows researchers to look for potential causes of those fixing actions.

In general the methodology is based on three main steps:

1. **Detection of fixing commits**: those are the commits that fix an issue in the source code management system.

2. **Identification of the lines that took place in a fixing commit**: those are the lines that were *touched* (*added*, *modified* or *removed*) in the commits detected as

fixing an issue in the source code.

3. **Detection of seeding commits**: those are the commits detected as causing the fixing action.

This methodology will be applied over the selected sample of projects from the Mozilla community. Those are found in the "root" directoy [1] of the Mozilla's repository. Although this community uses a specific technology when developing (the Mercurial source code management system), it can be extended to other communities using a SCM[2].

The structure of this chapter is as follows: section 3.1 introduces the basics to understand the source code management system. Besides, the commands to obtain the several pieces of information are detailed. Since the case of study uses Mercurial as its SCM system, specific peculiarities of this will be explained.

In second place, the retrieval process is detailed in section 3.2 where the tool *BlameMe* is introduced and the mining process detailed.

In third place, after detailing the data sources and tools that will be used, the general method that will be followed by all of the experiments is detailed in section 3.3. After this, specific peculiarities of each of the experiments undertaken are added to the general method in section 3.4.

Finally, a set of problems found during the retrieval process is specified, although the general validation of the tool and other considerations are specified in the Annexes section (chapter A).

3.1 Data Sources

In general terms, libre software projects provide a huge quantity of data related to software development. This data is publicly available to be used for anyone interested in mining them. The basic ones are those that help in the developing and maintenance stage of the project and help to coordinate the whole project ecosystem: source code management systems, mailing lists or bug tracking systems. In addition, there are others that also help in this process such as: releases of source code, wikis, web sites and some others.

This dissertation is focused on the analysis of the SCM that is the central point of all interactions, being a way of collaboration among all of the developers. Among others, the usual ones found in FLOSS projects are the CVS, Subversion, Git, Mercurial or Bazaar. The case of study, the Mozilla Foundation, uses Mercurial as its SCM. Initially this community used CVS, although it migrated to Mercurial around 2006.

[1] `http://hg.mozilla.org/`

[2] Further information about the threats to validity of the extension of the current methodology to other communities using different SCM is found at the *Threats to Validity* section (chapter 6, section 5.1).

The following subsection will introduce the SCMs and main pieces of information that are interesting from a researching point of view.

3.1.1 Introduction to the Source Code Management Systems

Source code management systems (SCMs in the following) are tools that help in the software development process. From the very beginning, FLOSS projects have used them as a way to centralize the work done by all the distributed people. As we have seen, FLOSS communities own some common characteristics, such as a huge distribution around the world, using different mechanisms to keep in touch.

So far, SCMs such as Subversion or CVS have been used as a way to support the development process. However, in recent years, some others have appeared trying to fill the gap left by the centralization of the source code development process and the distribution effort of FLOSS developers. All in all the main goal of this type of systems is to use them as a back-up strategy, having access to historical revisions of the source code. This let us to research them and study who is the author, what was added, modified or removed and when it was done.

In recent years, several communities have migrated to SCMs like Git or Mercurial. These migrations were due to several and specific characteristics that made them more attractive in terms of software development.

- Faster tools: This type of tools are in most of the cases faster than their older versions. Since they all work directly with the database found in the local disk, operations are much faster than by means of the network. This also facilitates to commit wherever a developer is located, even if she does not have access to the Internet.

- Decentralization: the previous characteristic facilitates the interaction with the local repository and it allows the developers to commit in almost any place. Thus, developers tend to commit more often and smaller changes.

- Being a decentralized SCM also means that there is always a working copy of the main branch being used by others. This is useful when the server hosting a centralized SCM fails. Using Git or Mercurial, each branch is distributed among all of the developers and probably, several backups have been made by the developers.

- Really cheap branches: the very existence of a clone of the repository is a new branch, although this would be merged later. In SVN or CVS, there are not clones, but checkouts. This means that later, a commit in the repository necessarily means a commit in the centralized repository.

- This type of SCMs eliminates the distinction between people with commit rights and people without those rights. Now, anyone could download a version of the repository and directly work on it. However, there still exist main branches maintained by the main developers which should receive the correspondent patch to include the changes.

In addition, there are specific advantages when using distributed SCMs from a researching point of view:

- As said before, having the database locally accessible is much faster to retrieve the data. In other centralized SCMs such as CVS or SVN the extraction of data is particulary slow due to the bottleneck represented by the network.
- The use of this type of SCM also offers new information. From a mining point of view, there is new data accessible [Bird *et al.*, 2009b] like the real authorship date. This is the example of people with no commit rights that had to create a patch and then submit those changes to a committer with commit rights. Finally, this developer could thank the participation by means of reflecting that in the log commit, in the AUTHORS file or in some other ways. Nowadays, and using some distributed SCM, that information could be specified in the same commit.
- The distributed SCMs also provide other types of new data and this is related to the time when the change to the source code was actually committed. In terms of a traditional SCM, the date registered in the log system is the date registered by the server when a submission was committed. In the case of a decentralized SCM, this information contains the real date (and timezone) when a submission was committed by the author and not the date when it was submitted to the server. In addition, when a server registered that date, this was registered in the timezone and the date of the server and not with the timezone and real date of the authors what creates incorrect data.

Finally, it is worth mentioning that each small piece of information found in any SCM is obtained thanks to the existence of specific tools provided by the SCM. An example of this is the use of the *hg log* or *hg diff* commands provided by the Mercurial repository. Those commands help to retrieve specific data needed in this dissertation and will be detailed in following subsections.

Understanding the log in Mercurial

The log provided by the command *hg log* is a summary of the commit undertook by a developer in a Mercurial repository. It shows the information related to the author, when the commit was submitted and the message written by the developer. This is commonly

used together with the *hg log -rXX -p* command which allows to see the message and also the changes done in the source code (where XX is a revision). An example of the outcome of this command is the following:

```
$ hg log -r11106
changeset:   11106:ce7b4d1c9978
user:        mozilla.mano@sent.com
date:        Thu Jan 31 10:05:22 2008 -0800
summary:     Bug 411088 - when deleting a tagged bookmark from the places organizer,
             the tag remains. r=dietrich, a=beltzner.
```

In addition, to the previous information, the *hg log -rXX -p* command would show information related to the differences between two versions. It is also plausible to be done by means of the *hg diff* as detailed in the following subsection.

3.1.2 Introduction to the *diff* Command

The *hg diff* command is based on the *diff* GNU util developed by Paul Eggert, Mike Haertel, David Hayes, Richard Stallman and Len Tower [MacKenzie *et al.*, 2002]. However there is an additional feature and this is that they are using the *diff* command with a special flag named as *XDF_NEED_MINIMAL* and helps to obtain the minimal diff file possible among all the options.

The basic diff algorithm is described in [Ukkonen, 1985], [Miller & Myers, 1985] and [Myers, 1986]. In any case, if no further options are detailed when running this command, the typical outcome will be the *unified diff*. This format is the usual one found in most of the SCMs what would help to extend this analysis.

Analyzing the *diff* used in Mercurial

A diff is a summary of the changes which were undertook, in this case, in a SCM between two files. The diff command compares the files line by line and summarizes the differences in a specific format. The following is an example of a diff between two commits directly taken from the mozilla-central repository (Firefox project).

```
$ hg diff -r11106 -r11107
diff -r ce7b4d1c9978 -r cbbdb92a2c31 browser/components/places/content/controller.js
--- a/browser/components/places/content/controller.js
Thu Jan 31 10:05:22 2008 -0800
```

```
+++ b/browser/components/places/content/controller.js
Thu Jan 31 10:10:40 2008 -0800
@@ -276,10 +276,14 @@
       if (nodes[i] == root)
         return false;

+      // Disallow removing shortcuts from the left pane
+      var nodeItemId = nodes[i].itemId;
+      if (PlacesUtils.annotations
+                     .itemHasAnnotation(nodeItemId, ORGANIZER_QUERY_ANNO))
+        return false;
+
       // Disallow removing the toolbar, menu and unfiled-bookmarks folders
-      var nodeItemId = nodes[i].itemId;
       if (!aIsMoveCommand &&
-          PlacesUtils.nodeIsFolder(nodes[i]) &&
           (nodeItemId == PlacesUtils.toolbarFolderId ||
            nodeItemId == PlacesUtils.unfiledBookmarksFolderId ||
            nodeItemId == PlacesUtils.bookmarksMenuFolderId))
```

The *hg diff* command, by default, shows the diff between two revisions using the unified format. This example shows specifically the last part of the *hg diff* command.

This format (or unidiff) is often used as input for the patch program (text needed when a patch is applied). There are two different types of diff formats, the unidiff (developed by Wayne Davison) and the one supported by the GNU Project, being the latter the chosen one as a standard to avoid arbitrary formatting of diffs.

The diff format starts with two-lines header where the original file name is preceded by "− − −" and the new file is preceded by "+++". After this, one or more change hunks (usually named as chunks) are specified, which contain information related to the differences in the file. Those lines that were added start with a "+" character, those removed start with a "-" character and those which were neither added, nor removed start with a space character " ". Finally, if a line is modified, this is represented as added and removed, so this changes will appear adjacent to one another. Thus, if a set of lines (all adjacent) are modified, the old version of the lines will appear starting all of them with a minus character and the new version of the lines will appear with a plus character.

The following example shows a diff output where two different files were touched. For each of them, there is a new set of headers that delimit the changes:

```
hg diff -r1110 -r1111
diff -r 9b9a54eca5a6 -r 646365171e09 content/base/src/nsGenericDOMDataNode.cpp
--- a/content/base/src/nsGenericDOMDataNode.cpp
Fri May 04 23:37:05 2007 -0700
+++ b/content/base/src/nsGenericDOMDataNode.cpp
Fri May 04 23:47:09 2007 -0700
@@ -567,6 +567,9 @@
                   (!aBindingParent && aParent &&
                    aParent->GetBindingParent() == GetBindingParent()),
                   "Already have a binding parent.  Unbind first!");
+  NS_PRECONDITION(aBindingParent != this || IsNativeAnonymous(),
+                  "Only native anonymous content should have itself as its "
+                  "own binding parent");

   if (!aBindingParent && aParent) {
     aBindingParent = aParent->GetBindingParent();
diff -r 9b9a54eca5a6 -r 646365171e09 content/base/src/nsGenericElement.cpp
--- a/content/base/src/nsGenericElement.cpp Fri May 04 23:37:05 2007 -0700
+++ b/content/base/src/nsGenericElement.cpp Fri May 04 23:47:09 2007 -0700
@@ -1756,6 +1756,9 @@
                   (!aBindingParent && aParent &&
                    aParent->GetBindingParent() == GetBindingParent()),
                   "Already have a binding parent.  Unbind first!");
+  NS_PRECONDITION(aBindingParent != this || IsNativeAnonymous(),
+                  "Only native anonymous content should have itself as its "
+                  "own binding parent");

   if (!aBindingParent && aParent) {
     aBindingParent = aParent->GetBindingParent();
```

In addition, each of the chunks begins with range information and later the set of files with the addition, deletions and other not handled lines. The range information format consists of two ranges surrounded by a double @ signs all just in one line. Next, there is a format example:

```
@@ -Range +Range @@
```

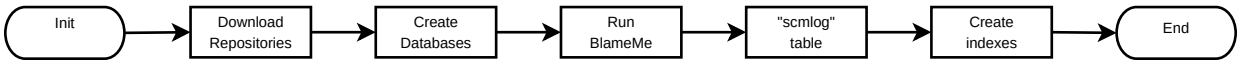

Figure 3.1: *Initial Step retrieving data: flowchart*

Thus, the chunk information is based on two ranges. The first one, preceded by a "-" character is the range for the original file and the second one preceded by a "+" character contains the information for the new file. Each of the *Ranges* is a subset of digits. The first one (let us name it as "l") indicates the starting line for the initial chunk. While the second number (let us name it as "s") is the number of lines that the chunk applies to for each respective file. Those values are always separated by a "," character. In the case that there is no *coma* the number by default for the "s" field is 1.

Hence, if it is needed to calculate the original file, it is necessary just to focus on the number of remaining and deleted lines and the "l" character. The sum of all of those lines will be the field "s" in the first range. With respect to the final file, the "s" value will be the same as adding all the added and remaining lines.

3.2 Retrieving data

Once the source code management system has been studied and all of the possible valuable pieces of information detected, the retrieval process is specified. This section shows the steps that have been followed in the initial process and are depicted in the flowchart at figure 3.1:

- **Download the repositories**: this first step downloads the whole set of repositories that later will be analyzed in the dissertation.

- **Create databases**: the databases must be created since the BlameMe tools expects to directly access them. The tables will be created by the several scripts and tools.

- **Run BlameMe**: BlameMe is the tool used for the diff parsing. This is run as many times as number of repositories studied.

- **Create *scmlog* table**: The scmlog table is created based on the log provided by the developers when committing a change.

- **Create indexe**s: four indexes are created to improve the performance of the tools and scripts that will be used later.

3.2.1 BlameMe: the diff Analyzer

BlameMe is the tool in charge of retrieving the differences between each pair of revisions for each of the analyzed projects. This tool is licensed under the GPLv3 and can be found at `http://git.libresoft.es` toolset.

Usage

The usage of the tool is specified in the following capture:

```
$ blameme --help

Usage: blameme [options]

It extracts data analyzing the differences between each pair of revisions
from a given project

Options:

   -h, --help           Print this usage message.
   -t, --type           Type of source code management system repository
                           (git|hg) Git or Mercurial (None)

Database output specific options (by default mysql is used as server):

   --db-user            Database user name (None)
   --db-password        Database user password (None)
   --db-database        Database name (None)
   --db-hostname        Name of the host where database server is running
                           (localhost)
   --db-port            Port where the database is (3306)
```

Since the output of the unified diff always follows the same structure, a states machine was built to control de whole retrieval process.

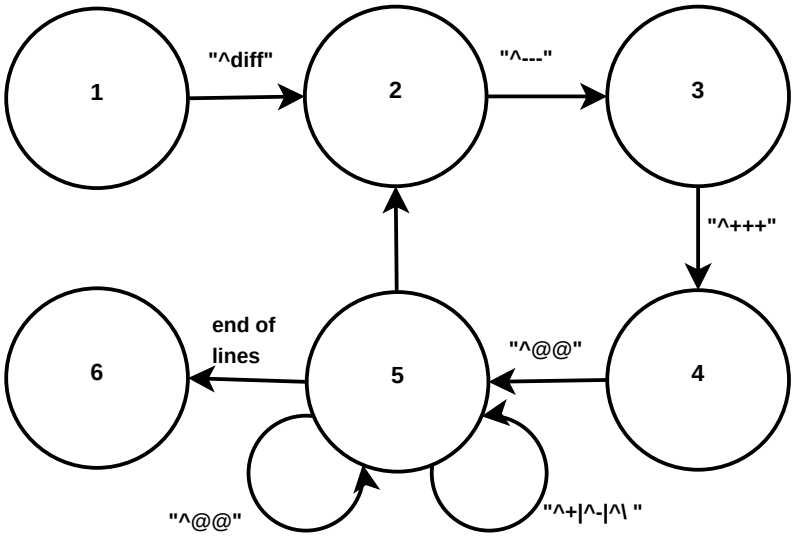

Figure 3.2: *BlameMe states machine*

States Machine

Figure 3.2 depicts this states machine based on the BlameMe source code. As can be seen, six states have been defined. This is extensible to other SCMs using the same output of their diff tools. The main tokens analyzed are each of the lines provided. And as previously explained, those are **diff**, , +++ to specify that this is a diff file. Then, one or more chunks of information are provided. Those start with the token **@@** and do not end till a new diff piece of information is found (going again to state number 2) or till the end of the file is found.

Thus, if tokens starting with **@@**, + or **-** are found, those are still chunks from the same diff piece of information.

Database Schema

The BlameMe tool creates two tables: **blame** and **files**. The addition of the **scmlog** table is done by means of a script. The resultant database can be seen in figure 3.3:

- **scmlog**: this table contains information from each of the commits: author, hash of the revision, original date, original timezone, date translated to +00:00 timezone and log message left by the developer.

- **blame**: this table contains all the changes retrieved for all the lines that took part in each pair of revisions. Here, it is possible to identify (with some limitations) the different lines and follow their life. Each line is associated to a file (*file_id*), has a specific state (ADDED, REMOVED or MODIFIED) and its original content.

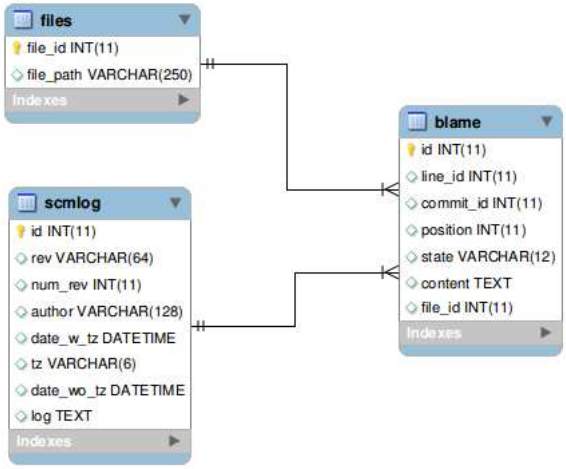

Figure 3.3: *BlameMe database schema*

Finally, each set of modifications are grouped by a *commit_id* which references the *scmlog* table.

- **files**: this table contains the actual path of the *file_id*.

3.3 General Overview of the methodology

In sections 3.1 and 3.2 the data sources that are used in the study, together with the basic technical infrastructure used were detailed. This section aims to describe the general method followed for the initial data mining approach. As described at the beginning of this chapter, the data mining process is based on three main steps: identification of commits fixing an issue (subsection 3.3.1), identification of the lines that were modified or removed (subsection 3.3.2) and identification, tracing backwards the history, up to the roots of that cause of the fixing action (subsection 3.3.3).

Regarding to the basic **variables** that will be used in this part of the methodology, those can be summarized in the following:

1. **Commits**: changes to the source code
   (a) **Bug seeding commit**: commits involuntary introducing an issue in the source code
   (b) **Bug fixing commit**: commits fixing issues in the source code
   (c) **Others**: rest of the commits
2. **Lines of source code**

3. **Author**: developer who owns the authorship of a change in the source code

4. **Time of the commit**: date when the commit was done

3.3.1 Detection of Commits Fixing an Issue

This section aims to detail how the fixing commits are detected and a later validation process of the heuristics used for the Mozilla community.

By definition, bug fixing commits are those commits that have fixed an issue in the source code. Since this dissertation is using publicly available data sources, this takes advantage of the traces left by the developers when committing changes. And this allows to study their activity from an academic point of view.

A bug fixing commit is that one whose log message follows a specific pattern: the key word *Bug* or *bug* followed by an integer. This integer is actually an identificator that references to the BTS (Bug Tracking System). As previously detailed in section 3.1.1, the log message is obtained by means of the *hg log* command. As an example the following output of the *hg log* command shows a commit fixing an issue:

```
$ hg log -r11106
changeset:   11106:ce7b4d1c9978
user:        mozilla.mano@sent.com
date:        Thu Jan 31 10:05:22 2008 -0800
summary:     Bug 411088 - when deleting a tagged bookmark from the places organizer,
             the tag remains. r=dietrich, a=beltzner.
```

Besides, that issue is identified by the integer *411088* and this references to the Mozilla bug tracking system[3].

As seen, the log message provides information such as:

- **changeset**: this field provides the "hash" values of the commit.
- **user**: this provides information from the user that committed the changes. In some cases, the email address and the user name are provided. Through email, information about companies participating in the community could be infered.
- **date**: this date, since Mercurial is a distributed SCM, is the local time when the commit was done by the *user*. Thus, the date is not kept centralized like in other SCMs such as CVS or SVN. The date is the actual time when the commit was uploaded, using the timezone of the developer and not the server.

[3]Specifically to the issue found at the URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=411088`

- **summary**: this provides information regarding to the type of changes done to the source code. In some communities, there are specific policies with respect to the type of message that can be left in this field. In the case of the Mozilla community, each change, if this intended to fix an issue, (or at least if this is related to the BTS), starts with the key word *Bug* or *bug* followed by an integer that points to a bug tracking system report.

The detection of bug fixing commits is done thanks to the last field: *summary* that allows to check if a commit is fixing an issue or is not. This approach has also been used by other authors [Kim *et al.*, 2008d]. In addition, a validation process of the selection of the bug fixing commits is done in the following subsection.

Validation of the heuristics used when discovering bug-fixing commits

In order to validate the empirical approach of defining what a *bug fixing commit* is, this subsection reports on the log messages provided by one of the studied projects (*comm-central*), and this evaluates the consistency and reliability of their records with regards to bug fixing activities. To achieve this purpose, an empirical approach was developed and then checked the number of false positives and false negatives obtained. The approach used is as follows: at first, 100 commits were randomly selected. From each of them, an initial binary classification was done: a commit is fixing an issue or is not. Later, based on that classification, it was checked manually if these are actually bugs or not, either checking the underlying source code or by parsing the relative Bug Tracking System). The results for evaluating the precision and recall of such approximation, and its constituent parts are as follows:

```
(TP) True positive:  78
(FP) False positive:  7
(TN) True negatives:  6
(FN) False negatives:  9
Total commits:  100
```

Therefore we evaluated:

- Positive predictive value: $TP/(TP + FP) = 78/(78 + 7) = 91,7\%$
- Negative predictive value: $TN/(FN + TN) = 40\%$
- Sensitivity $= TP/(TP + FN) = 78/78 + 9 = 89,65\%$
- Specificity $= TN/(FP + TN) = 6/7 + 6 = 46,15\%$

Since the *Precision* actually coincides with the positive predictive, and the *Recall* coincides with the sensitivity, we conclude that $precision = 91,7\%$ and $recall = 89,65\%$.

3.3.2 Detecting Lines

This section aims to explain in a more detailed way the differences between *bug seeding* and *bug fixing* commits. Using the aforementioned definitions as a starting point, the bug seeding and bug fixing commits are a subset of commits from the overall activity within a given SCM. Bug fixing commits have been obtained by studying the logs left by committers in the SCM. Specifically, those that show the tuple "bug" + "ID" being bug a keyword and ID an integer which references the BTS. Thus, given a commit and the output of the *hg diff* command, it is possible to obtain a complete picture of the lines that were *added*, *modified* or *removed*, but also the developer, the date and the handled files. In addition to the previous examples detailed at section 3.1.2, the following is another similar example that will help to illustrate again the output of the *hg log* and *hg diff* commands.

```
$ hg log -r11106
changeset:   11106:ce7b4d1c9978
user:        mozilla.mano@sent.com
date:        Thu Jan 31 10:05:22 2008 -0800
summary:     Bug 411088 - when deleting a tagged bookmark from the places organizer,
             the tag remains. r=dietrich, a=beltzner.

$ hg diff -r11106 -r11107
diff -r ce7b4d1c9978 -r cbbdb92a2c31 browser/components/places/content/controller.js
--- a/browser/components/places/content/controller.js
Thu Jan 31 10:05:22 2008 -0800
+++ b/browser/components/places/content/controller.js
Thu Jan 31 10:10:40 2008 -0800
@@ -276,10 +276,14 @@
       if (nodes[i] == root)
         return false;

+      // Disallow removing shortcuts from the left pane
+      var nodeItemId = nodes[i].itemId;
+      if (PlacesUtils.annotations
+                     .itemHasAnnotation(nodeItemId, ORGANIZER_QUERY_ANNO))
+        return false;
+
       // Disallow removing the toolbar, menu and unfiled-bookmarks folders
-      var nodeItemId = nodes[i].itemId;
```

```
       if (!aIsMoveCommand &&
-           PlacesUtils.nodeIsFolder(nodes[i]) &&
            (nodeItemId == PlacesUtils.toolbarFolderId ||
             nodeItemId == PlacesUtils.unfiledBookmarksFolderId ||
             nodeItemId == PlacesUtils.bookmarksMenuFolderId))
```

In this example, the above output represents the removal of two lines (indicated with a "-" sign at the beginning of a line) and the addition of six lines (indicated with a "+" at the beginning of the line). This is also an example of a change deriving from a bug-fixing commit (i.e., a commit whose log message follows the regular expression: $(b|B)ug[1-9]+.*$). As it is detailed in the following subsection, the process of looking for the *root* causes of the bug is based on the tracking of the lines that were *modified* or *removed* in a fixing commit. This assumption is based on the assumption made by previous literature such as [Śliwerski *et al.*, 2005a] and also used in [Kim *et al.*, 2008a].

Summarizing: the suspicious lines of *causing the fixing action* in the previous example are the two that were *removed*. And those are detected using the log message left by the developer and the information provided by the output of the *hg diff* command line.

3.3.3 Detection of Commits Seeding Issues

The final step consists of tracing the life of each of the lines (*modified* or *removed*) involved in a bug fixing commit. Hence, in the empirical analysis conducted in this dissertation, and using the information provided by the *hg diff* tool, all the removed and modified lines are traced backwards in the repository, and it has been noted when each line has been previously added or modified. It is worth mentioning that a bug fixing commit has *by definition* at least one previous bug seeding commit where all the lines were previously *handled* or *touched*.

Based on various scenarios, such set of lines could have been added or modified in the *same* commit (and by just one committer) or in *different* previous commits. In the latter, such lines could be committed by the same committer, or by different committers. It should be noted that in a given repository there are at least the same number of bug seeding and bug fixing commits and that a bug fixing commit may become a potential bug seeding commit.

As a summary and previously specified, the main assumption of this algorithm is based on the set of lines handled in a bug fixing commit: if one or all the lines of that set of lines have been *modified* or *removed*, those are at risk of being part of the problem. The analysis of when such lines *seeded* the bug is based on the immediately previous change (either addition or modification). This assumption, as an example was used to mark classify

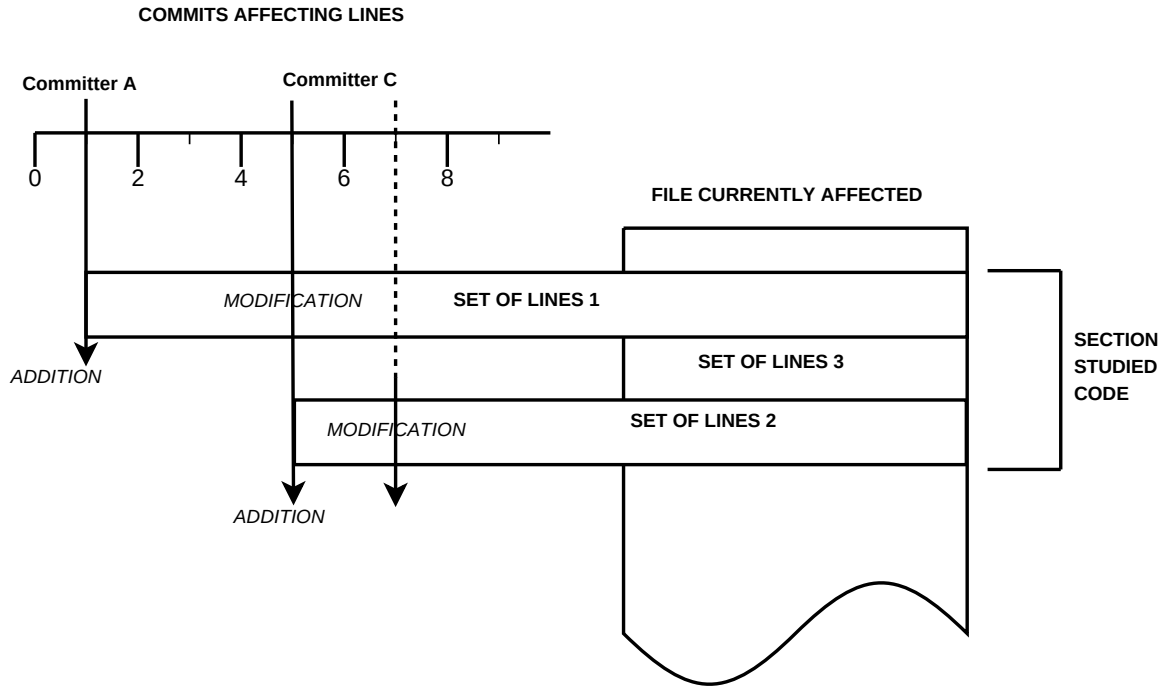

Figure 3.4: *Detecting the time when a bug seeding commit happened.*

changes into *cleany* or *buggy* [Kim *et al.*, 2008a], [Śliwerski *et al.*, 2005a].

A visual example is depicted in figure 3.4 where a commit is detected as seeding an error in the source code. For the section of studied code (at the right side of the figure), there are three sets of lines that were handled (set of lines 1, 2 and 3). In the case of the first one, that was added in commit number 1 by committer A and later modified in commit number 5 by Committer C. The set of lines 2 was added by committer C in commit number 5 and later modified in commit number 7 by the same committer.

Thus, using the assumption that the lines that are modified or removed in the commit that fixed an issue are now at risk of causing that fixing action, those are trace back to their previous commit. In the case of the set of lines 1, those were previously modified by committer C in the commit number 5. With respect to the set of lines 2, those were modified in commit number 7 by committer C. Finally, regarding to the set of lines 3, those were not affected in the commit that fixed the bug, so they are meaningless in this example.

Thus, the bug seeding commits were in this case, the commit number 5, since there the set of lines 1 were modified. And the commit number 7, since the set of lines 2 were modified. In all of the cases, the developer that modified the lines at risk of being causing the fixing action was the committer C.

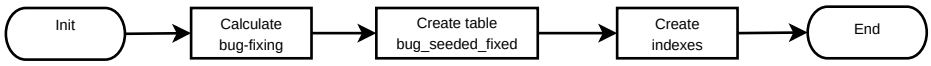

Figure 3.5: *Second step retrieving data: flowchart*

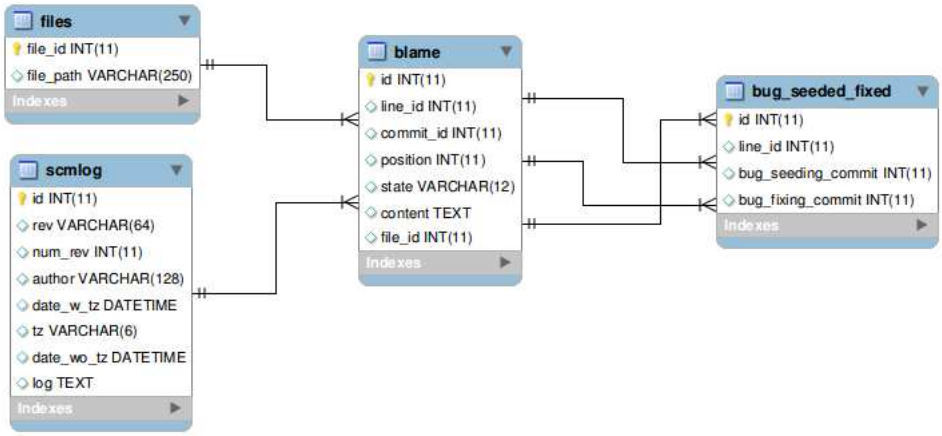

Figure 3.6: *Resultant database schema*

Retrieving seeding and fixing commits

This subsection aims to explain the technical approach followed to store in the database those commits that have fixed an issue and their previous bug seeding commits. The following steps were done (also depicted in figure 3.5).

- **Calculate bug fixing commits**: those are calculated as aforementioned, using the heuristic: $(b|B)ug[1-9]+.*$.
- **Create "bug_seeded_fixed" table**: this table is calculated in order to obtain only information about the fixing commit, the lines involved, and their previous bug-seeding commits.
- **Create indexes**: those are created to improve the performance of the scripts.

The resultant database schema is shown in figure 3.6, where the table *bug_seeded_fixed* is added:

- **bug_seeded_fixed**: this table contains specific information for each set of bug-seeding commits - bug-fixing commit. For each commit detected as fixing an issue (*bug_seeding_commit*), its lines are recovered. For each line (*line_id*), it is calculated the exactly previous step, or in other words, the commit when it was added or modified and detected as being a bug-seeding commit (*bug_seeding_commit*).

Finally, it is worth mentioning that the retrieval process is done following the chronological life of the repository. This allows to store, for each pair of revisions that

are calculated, the files that were added, modified or removed (also their lines) and consequently, update, for each revision the tree of files of the repository. Thus, while the process of storing the data in the database follows the life of the project (from the first commit to the very last one), the process of calculating the seeding commits follows a backwards path: for a given commit, its lines are calculated and later the seeding commits.

3.4 Specific Methodology

This section aims to detail extra methods used for specific analysis along this thesis. For all of them, the basic structure of how bug fixing and bug seeding commits are retrieved is the same and has been already detailed. However, depending on the experiment, specific tables will be added to the database schema. In addition, specific analysis will be done using external tools, as in the case of retrieving information from the BTS or other similar cases.

As seen in chapter 1 and 2, the general analysis is divided into two main sections: the study of the bug life cycle and the potential causes of bug introduction actions.

With this respect, subsection 3.4.1 explains the method for filtering the dataset based on the study of the programming languages.

In second place, the peculiarities of the analysis of the bug life cycle are detailed in subsection 3.4.2. There, all of the studies related to the calculation of the time to fix a bug considering the SCM and BTS are detailed. In addition, subsection 3.4.3 will specify how the demographic studies were done. This will help in the estimation of the bugs finding rate per project.

Regarding to the human factors that may alter the bug seeding rate, subsection 3.4.4 will detail the way experience or expertise is measured. In addition, subsection 3.4.5 will detail how the characterization of developers is carried out by means of the type of activity committed to the SCM. And finally, subsection 3.4.6 will explain the method followed by the study of the time of the day.

3.4.1 Programming Languages Considered in the Study

The Mozilla Foundation presents specific peculiarities that makes this project special if considered to traditional software engineering studies. In most of the literature, it has been considered programming languages such as C, Java, C++ or others as the bases of the studies. However, marked up source code such as HTML or XML tend to be not considered as proper source code. In fact, in some occasions these files are automatically created by some tools.

For instance, there are specific tools for Java or other programming languages where the documentation is automatically created based in the comments left by the developers

and these files are in overall HTML.

However, in this dissertation it is necessary to open this discussion since a big part of the source code is written in XUL (XML User Interface Language), what is a type of improved XML used to create the user interface applications. Thus, it is necessary to deepen more in this study and check the distribution of bug seeding and fixing commits in the overall distribution of programming languages.

This will help, in the end, to filter the initial dataset of commits, only taking those with the specific programming languages needed.

The **variables** used will be only the **bug seeding** and **bug fixing commits**, limiting its quantity by filtering the dataset by programming language.

Studied Programming Languages

The retrieval process of the programming languages is initially done by the BlameMe tool. Those are pre-selected by means of a regular expression and filtered by the extension found in each of the files provided by the *hg diff* output.

```
extensions = ".+\.(java|c|sh|py|h|am|cc|sql|rb|perl|
              tcl|el|js|css|xul|cpp|idl|xpm|xml|xhtml|html|dtd)$"
```

Later, the programming languages are stored in the database of BlameMe. As seen in the schema (figure 3.3) there are three main tables: *scmlog*, *blame* and *files*. The latter consists of two fields: *file_id* and *file_path*, being the second one the place where the name and extension of the file is stored.

Hence, as previously commented, along the source code of the Mozilla community it is possible to find *traditional* programming languages such as Java, C or C++, *scripting* languages such as Python or Perl and *marked up* languages such as XUL, XML or HTML. However, it is not clear if the last set of programming languages behaves in the same way as more traditional programming languages do.

In order to facilitate this process, a new table is created with specific information about the programming languages and the bug seeding ratio obtained. This table is calculated using information from the *blame* and it is named as *files_per_commit*. This will contain a match between the *commit_id* found at the *scmlog* table and the *file_id* found in table *files*.

3.4.2 Time to Fix a Bug

This subsection aims to show the method followed to study the time to fix a bug. This outcome is obtained by subtracting the final time when an issue has been fixed in a bug fixing commit from the time when the bug seeding commit took place.

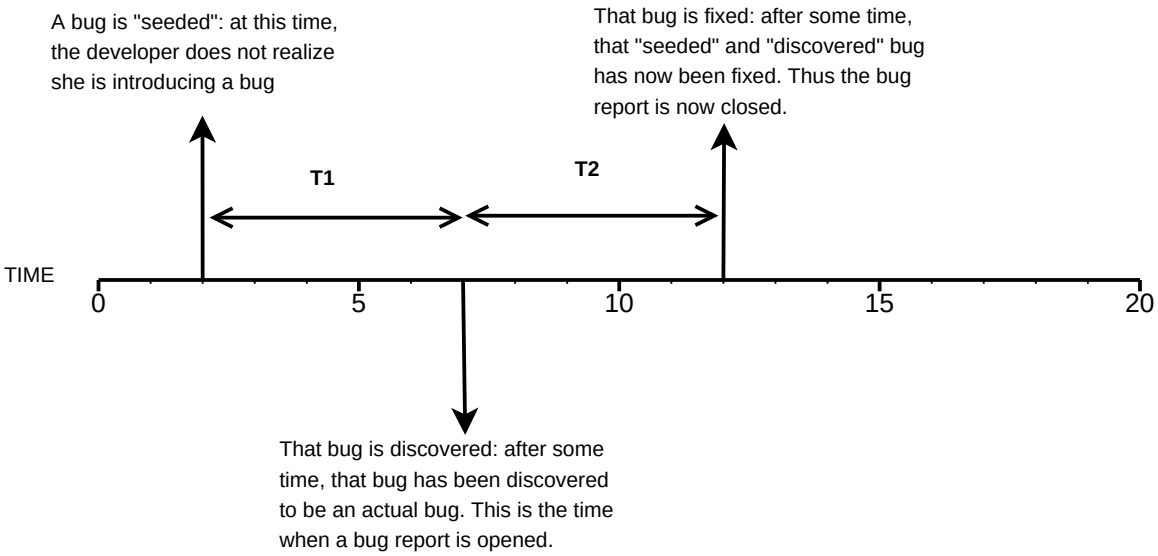

Figure 3.7: *Time to fix a bug diagram*

In addition, specific information is retrieved from the bug tracking system in order to study the differences in time between the four following variables:

1. **Bug fixing commit time**: date when the bug fixing commit takes place
2. **Bug seeding commit time**: date when the bug seeding commit takes place (when the issue is involuntary introduced in the source code)
3. **Open report time**: date when the bug report is opened
4. **Close report time**: date when the bug report is closed

Although, it can be seen as similar the bug seeding and the open report events, there are main differences. Actually, the open report time is the event when the error has been detected by an user or a developer. However, the time when a commit is seeded is the time when the error was involuntary introduced in the source code. Thus, from an intuitive point of view, the time when an error was introduced in the source code should appear before the time when the bug report is opened. The time of each event is also depicted in figure 3.7.

Thus, there are three main states for a bug:

1. **A bug is "seeded"**: this is the initial state of the bug. A developer may have introduced a bug, however, it has not been detected yet. Potentially, all of the commits are future bugs, however, a subset of those commits become bug seeding commits.

2. **A bug is "discovered"**: when the bug has been discovered, a report is open in the bug tracking system. At this point, the bug enters in the life cycle of the bug tracking system and the developers have realized about its existence.

3. **A bug is "fixed"**: this is the last step in the life of a bug. In some cases a bug may be fixed or not. However, **this study is focused only on those which are finally fixed**. In addition, when a bug is "fixed", it has been located in both places: at the commit message, and at the bug tracking system.

The T1 time in figure 3.7 represents the time between the two first states, or in other words, between the time when this is seeded, and the time when this is actually discovered. The T2 time is the time between the opening and closing of the bug report in the bug tracking system. So far, the T2 time has been studied from several perspectives, however **the T1 and T1 + T2 time are still open research question**.

Finally, it is necessary to mention that the set of specific bugs that have been retrieved are those specified in the commit log, and so on, those that have been detected as fixing a bug. Thus, the method followed in this case consists of accessing the *scmlog* where the log message is stored and retrieve the specific bug id. With that set of identifiers, now they are the input of *Bicho* to be specifically retrieved from the Mozilla's Bugzilla.

Retrieval Process

For this purpose, two new tables are needed. In first place, the *seeded_fixed_couple* table, which only contains the commit identification from each couple of bug seeding/bug fixing commits found in the *bug_seeded_fixed* table. This table is only used during a temporal time, thus, later will be deleted.

It was also necessary to create the *time_diff* table where the differences in time between those pair of commits is obtained. In this case, the indexes must be created before the creation of the *time_diff* table since they are needed for a better performance.

- **Create seeded_fixed_couple table**: this table is temporary and contains the couple bug seeding, bug fixing commit.

- **Create indexes**: those must be created before the following table is created in order to improve the performance of the scripts.

- **Create time_diff table**: this table will contain the differences in time between each couple of seeded and fixed commits.

Figure 3.8 shows the flowchart that is followed to create these two new tables.

The resultant database schema is shown in figure 3.9, where *seeded_fixed_couple* and *time_diff* tables are added::

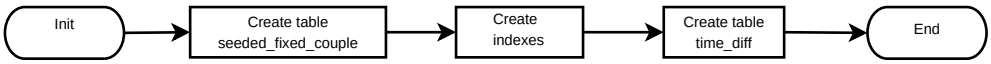

Figure 3.8: *Third step retrieving data: flowchart*

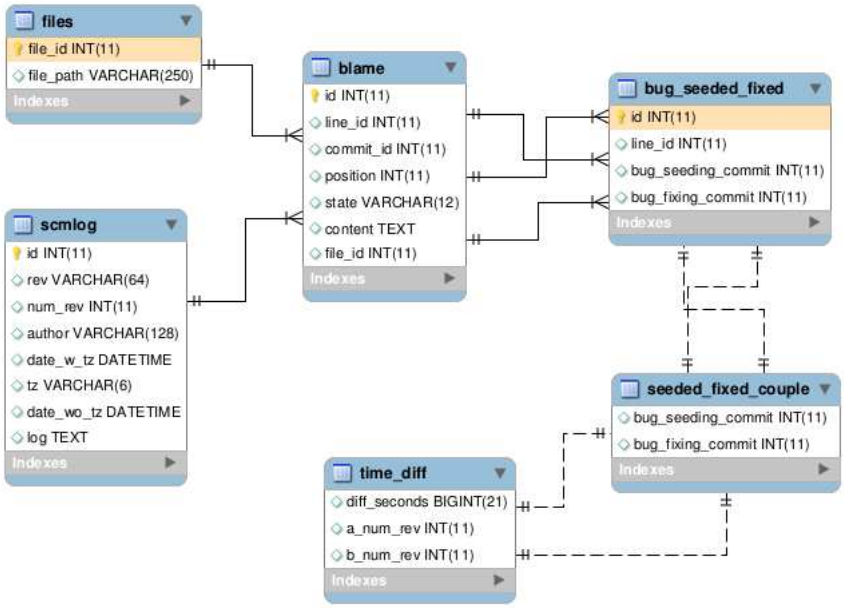

Figure 3.9: *Resultant database schema*

- **seeded_fixed_couple**: this table contains specific information for each set of bug seeding commit - bug fixing commit. Table *bug_seeded_fixed*, for each line, their associated bug-seeding and bug-fixing commits are also stored. In this table, only the couple bug-seeding and bug-fixing commit is stored. Thus, this table is ignoring the number of lines per commit.

- **time_diff**: this table contains the difference, measured in seconds, between a bug-fixing commit and all of its associated bug-seeding commits. Thus, for a given bug-fixing commit (**b_num_rev**), this will appear at least once, related to a bug-seeding commit (**a_num_rev**) and with the difference in time between both commits (**diff_seconds**). This data is obtained calculating the difference between two commits, but calculating all of the dates with respect to the GMT +0. This avoid the difficulties related to the time zone of each developer. Finally, it is worth mentioning that this data is strictly obtained from the *scmlog* table.

Retrieving Bugzilla Information

In addition to the previous retrieval process focused on the SCM, this subsection aims to describe the retrieval process focused on the BTS. Indeed, this is the traditional way of

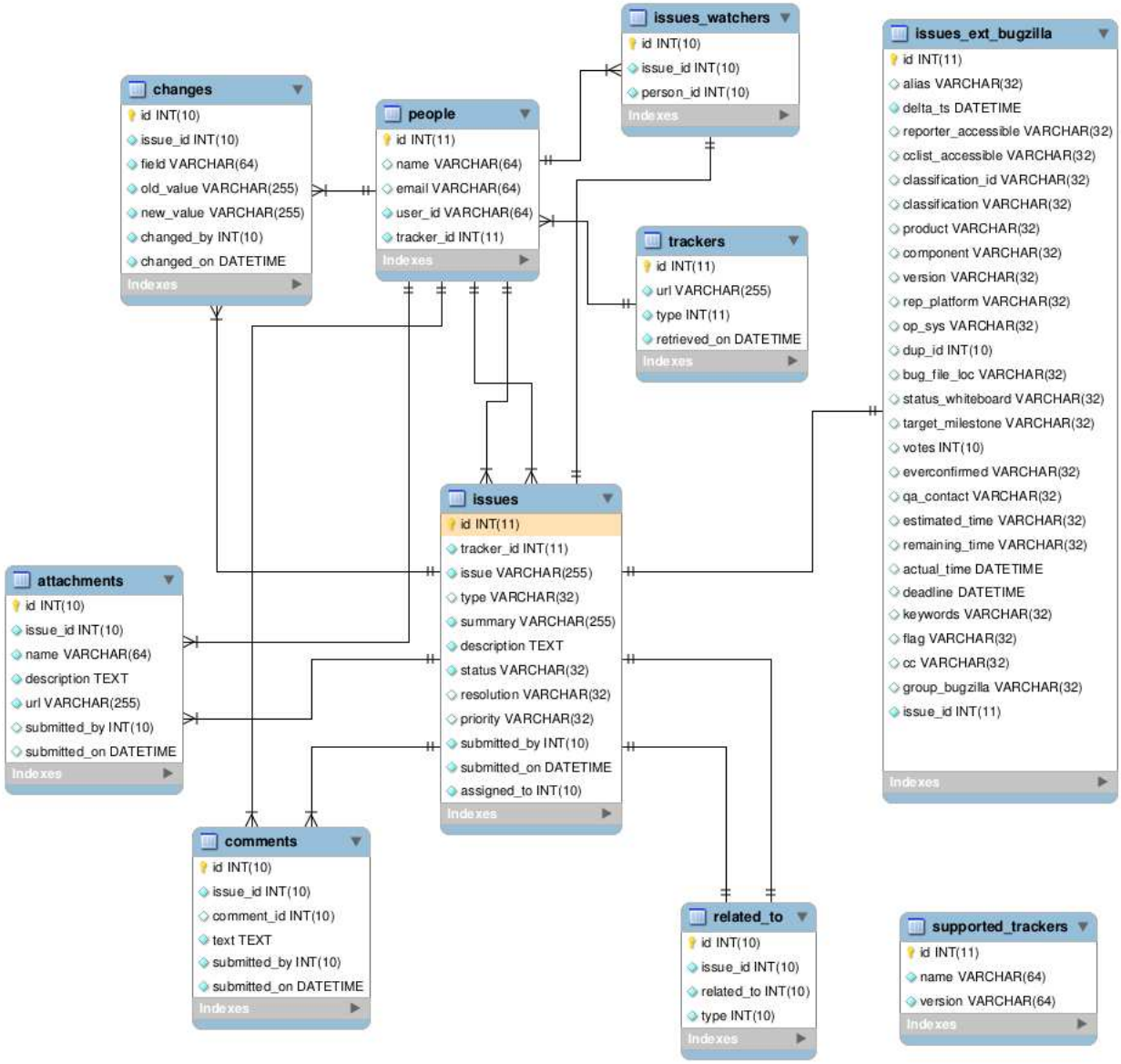

Figure 3.10: *Bicho database schema*

calculating the time to fix a bug as seen in the literature (chapter 2, section 2.2.1). For this purpose, the tool *Bicho* was created [Gregorio Robles & Herraiz, 2011].

Nowadays, this tool is being improved and is continuously evolving, Its goal consists of retrieving information from a bug tracking system and create a relational database. So far, it is compatible with the Bugzilla of KDE, GNOME and Apache. In addition, it also offers support for Jira and SourceForge.

The database schema is depicted in figure 3.10 where the most important tables in this study are the following:

- **issues**: this table contains the basic information for a given bug, such as the

submitter, the current state, the resolution and others.

- **changes**: this table contains each of the modifications to any of the fields associated to a bug report. Thus, with this table, and the specific state: CLOSED; OPEN or others, it is easy to associate the date when that change happened.

The study of the time to fix a bug is found in two different databases the information for a given bug: the SCM information found at the database created by *BlameMe* and the BTS information found at the database created by *Bicho*. In order to link both data sources, a new table *date_analysis* was created where for each bug detected at the SCM, it is obtained by means of *Bicho* the time of opening and closing report.

3.4.3 Demographical Study of the Issues

Demographic studies (figure 3.11) are usually created to study the ages of specific populations in a given time and for a given set of people or other actors. Those are also named as pyramids of population.

The demographical study aims to detail how the bug fixing activity is being carried out and checking for how long tend to remain in the source code the *seeded* issues.

The variables that will be used in this experiment are the **bug seeding commits**, their **age** measured in months and the **bug fixing commits**. The demographic studies will show the remaining bugs in the source code and their age.

In addition, this method aims to detail the process of understanding how fast is the finding-bugs process. This could be useful to make estimations of remaining bugs from a given date and how the finding process takes place. It is expected that the number of bugs decreases pretty quick at the beginning while few of them will remain for longer time.

This demographic study will be implemented as calculating bugs by groups of ages. The interval is measured in one month and this is calculated for all of the projects in the cases of study. The followed method consists of calculating, for a given month (t), all of the commits that were detecting as *seeding* a bug and studying its evolution. In the following month (t+1), those are the same number of seeding bugs remaining (those that have not been already fixed) minus those that were fixed (given by the bug fixing commit).

In other words, the y-axis in the chart represents the age of the bug (not still fixed) in the community. And the x-axis represents the number of remaining bugs still to be discovered.

By definition of bug seeding commit, this is always associated to a bug fixing commit. This is similar to the populations in real life: for a given set of people that were born in a given year, their number will not be increased. Thus, they are always a decremental function.

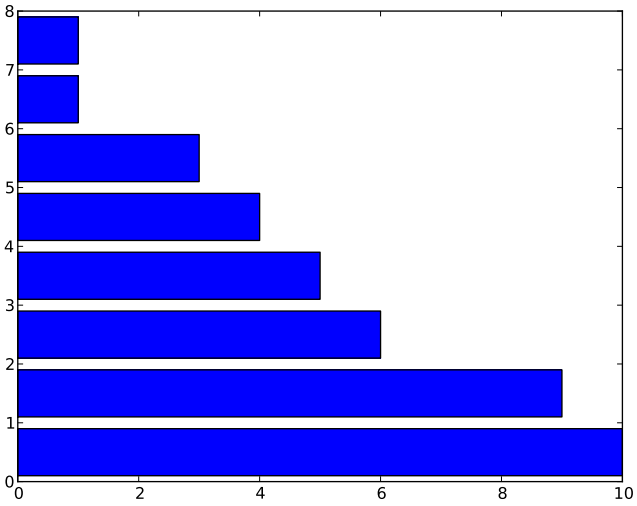

Figure 3.11: *Example of demographic study: the y-axis represents the age of the bugs in months and the x-axis the number of remaining bugs.*

These pyramids of population are interesting from a descriptive point of view, where it can visually seen how is a typical population of remaining bugs to be discovered. In addition, the evolution of the pyramids of population will help to determine the estimation of how fast the bugs are being found by developers.

3.4.4 Familiarity with the Source Code

The goal of this subsection consists of analyzing the experience of the developers compared to the usual "quality" of the changes done in the source code. For this purpose, new variables are introduced and analyzed:

1. **Bug seeding ratio**: this is the number of "buggy" changes out of the total activity (measured for a developer or for a project).

2. **Territoriality**: this metrics has been previously used by other authors [German, 2004a], [Robles *et al.*, 2006], [Robles, 2006b] and measured the number of files that are owned by just one developer. This is also applicable to other artifacts in the source code such as methods, classes or modules.

3. **Experience**: Depending on the author, experience can be measured in a simple way as the difference in time between the first and last commit, the number of commits, the traces left in the community (like in mailing lists or BTS') and others. Along this dissertation experience is measured in number of commits, number of fixing commits and territoriality. In any case, experience is a subjective concept and

measuring experience is not a trivial task. In fact, the very definition of experience is not a trivial task.

However, in this analysis, the goal is to check if there is any relationship between the total life in a project, and the number of *buggy* changes in the source code. From an intuitive perspective, the longer a developer has worked in a project, the higher her expertise (even more if tacit knowledge is referred) and the lower her bug seeding ratio.

3.4.5 Characterization of Developers

Based on previous variables, like **bug seeding** and **bug fixing commits**, those commits allow to study from another perspective the type of activity carried out by the developers. It has been observed how some developers tend to focus their productivity on maintenance tasks (fixing errors), others fixing errors and other tasks (that later are detected to be *buggy*) and finally, others that barely do any task related to maintenance activities (where no bug fixing commits are detected).

This raises specific questions related to the potential characterization of developers based on this approach.

It is worth to bring in context the concept of role in FLOSS communities. As detailed in [Fogel, 2005] and [Bacon, 2009], there are different roles played by different members. Even more, some developers may play several roles (translation, developer, documentation and others). Thus, our first approach is to distinguish those developers with a higher ratio of bug seeding commits and those with a higher ratio of bug fixing commits.

If we take the extremes of those two sets of people, we will find people with a quite high ratio of bug fixing commits, but close to 0 ratio of bug seeding commits. These could be classified as potential *pure* maintainers. On the other hand, those with a quite high bug seeding ratio and quite low bug fixing ratio may be seen as a people who do not work with the bug tracking system, but develops during their whole time participating in the community.

The method followed to classify developers is based on the following approach:

1. Number of commits per developer
2. Number of bug seeding commits per developer
3. Number of bug fixing commits per developers
4. The previous sets of commits are aggregated and the regression line obtained as an approximation (it has been observed quite high correlation of the resultant charts and the lines). In addition, the resultant function is based on the slope (m) and there is no constant in the equation. Thus: $f(x) = m * x$.

5. Both aggregation of data (bug-fixing commits (*bfc*) and bug-seeding commits (*bsc*)) follow the same behavior. Thus, there are two different slopes and functions defined: $f(x)_{bsc} = m_{bsc} * x$ and $f(x)_{bfc} = m_{bfc} * x$.

6. In order to obtain the difference of the areas, the integration of the functions is defined and limited by the number of dots studied (from 0 to the length of the array of dots): $f(x) = \int_0^{len} m * xdt$

7. $f(x) = \int_0^{len_{bsc}} m_{bsc} * xdt - \int_0^{len_{bfc}} m_{bfc} * xdt$

8. Finally, the differences are studied and a potential classification of developers is obtained.

3.4.6 Timeslots Analysis

The final sets of experiments are all based on a timeslot analysis aggregating the values in a 24 hours timeframe. From this perspective, the following variables are studied:

1. **Time of the day**: the day is divided in 24 hours

2. **Bug seeding ratio**: number of bug seeding commits out of the number of commits

3. **Commits**

4. **Core authors**: set of developers (around a 20% of them) that usually undertake a 80% of the total changes in the source code

5. **Non-core authors**: rest of the 80% of the developers

6. **Comfort time**: time when a developers usually work. In some cases, for external factors, a developer has to work out of her comfort time.

In addition, it is necessary to mention the major assumptions done for this experiment that help to simplify the analysis: **all of the days are taken as being the same**. Since the data is aggregated, we do not care if a specific set of days are bank holidays or other similar cases.

Retrieving Data

The retrieval process is based on the time of the day when a change in the source code is done. This information has already been included in previous steps of the general method. Only, the division between core and non-core committers is added to the general database schema.

For this purpose, two tables are added to help in the retrieval process:

- **core_authors**: this table contains the name of each of the authors that have been detected as *core*, in an aggregated way for the whole history of the project. This also contains the number of commits for each developer.

- **not_core_authors**: this table contains the name of each of the authors that have been detected as not being part of the core group, in an aggregated way for the whole history of the project. This also contains the number of commits per developer.

Chapter 4

Analysis

"To err is human–and to blame it on a computer is even more so"
Robert Orben - Magician

This chapter presents the applicability of the methodology detailed in chapter 3. All of the following analysis have been obtained analyzing the publicly available data sources from the Mozilla community and can be partially extended to others. As a reminder, the goal of this thesis is to study the bug life cycle and later to study the potential causes of seeding errors in the source code due to human related factors.

This chapter is divided into sections. Each of them aims to study specific parts of the initial research goals. The first of them, section 4.1, is an introductory analysis to the case of study where several projects chosen from the Mozilla Foundation are studied. One of the peculiarities of this community is the definition of a *general policy*. This stresses the point of *core* projects that requires extra effort from developers to double check the changes to the source code. This is expected to provide an extra review that will help in the maintenance process.

Section 4.2 details several known issues that may affect the quality of the dataset and specifies how this is filtered. Among others, it is worth mentioning the programming languages that are being analyzed, such as C or C++, while others have been removed from the dataset, such as XUL or XML[1].

Bug seeding activity is detailed in sections 4.3, 4.4 and 4.5. With respect to the first one, the *time to fix a bug* in the source code is studied from different perspectives and compared to the time to fix a bug in the bug tracking system. There is a main difference between these two times: the first one takes into account the time when a bug was *seeded*, while the second one takes into account the time when a bug is *detected*. In

[1]Those have been removed since the development process may be helped by external tools using a GUI interface. In any case, they are analyzed in this section.

section 4.4, a *demographical study* is done in order to study the life of the population of bugs. Besides, the *bug finding rate* is analyzed and compared among projects in order to look for a potential model of estimating fixing bugs. Finally, in section 4.5 the *defective fixing changes* are analyzed together with a study of the typical number of bug seeding commits associated to a bug fixing commit.

In section 4.6, the concept of *bug seeding ratio* is introduced (being the ratio between the bug seeding commits and the total number of commits) and its evolution analyzed at the granularity of projects (subsection 4.6.1), at the granularity of developers plus a *characterization* of their activities (subsection 4.6.3) and using those concepts to study the correlation between experience and bug seeding ratio (subsection 4.6.2).

Finally, a *timeslot analysis* is done to study the relationship between the time of the day and the potential errors found in the source code (sections 4.7). In first place, the aggregated history of the communities is studied in order to observe how the *effort* is distributed along a 24 hours framework. This helps to compare the distributions of resultant effort. In addition the bug seeding ratio is analyzed through the 24 hours checking those timeframes where there are more possibilities that an error is introduced in the source code (section 4.7.2). In second place, other human factors are studied. Again, the concept of experience is introduced, but this time using the definition of *core* and *non-core* committer during the last year of history. Finally, the *comfort* timeframe concept is introduced and this is the time where committers usually work. Having in mind this concept, the most important developers (measuring in commits) of the community are studied. And more specifically their timeframes of activity.

4.1 Case of Study: The Mozilla Community

The case of study is focused on the Mozilla community. Although there exist other FLOSS communities, the selection of this community was done for the following reasons:

1. The use of *Mercurial* as the source code management system. To the best of our knowledge, there are not FLOSS Mercurial analyzers. This implies the creation of a new tool to retrieve the information stored in this SCM. In addition, the Mercurial repository has got specific peculiarities as this is used by the Mozilla community and as it is:

    - **Research friendly**: this is probably the key factor to select this community. The repositories have been previously studied by other authors. The validation process used in this dissertation has corroborated the use of the SCM as a trustful data source when looking for fixing information. In addition, the policy

of adding the bug information has helped when linking two data sources: SCM and BTS.

- **Local time**: this is also a key factor. The last set of experiments are based on the analysis of a 24 hours framework. The option of having the local time, have allowed to study this type of behavior from the Mozilla community. This is also applicable to other distributed systems, such as Git or Bazaar.
- There is **no difference between author and committer**. All of the developers are authors. This helps to reach the real distribution of effort in the community. In other cases, the authors need to submit a patch to the committers, that are the ones who have the right to commit changes in the SCM.
- **Migration of the history** from scratch: although this could be seen as a potential lack of data, this is not an issue. Having in mind that the analysis of the dissertation is not focused on the evolution of different artifacts (at least measuring evolution from scratch), but focused on specific timeframes, this methodology could have been applied with less or more history.

2. **The openness of the data set**: all of the data sources are publicly available. This will help in the replicability process of the experiments undertaken in this dissertation.

Regarding to the community itself and according to the mission specified at their website[2], the Mozilla Foundation is a non-profit organization dedicated to foster the "openness, innovation and opportunity on the web" [3]. This community consists of several projects: web browser, email client and other tools that help in the web development process. All of these projects are listed in the following subsection.

4.1.1 Projects Analyzed

The analysis is based on the Mercurial repository [4] which offers a list of repositories that can be easily *cloned* (or downloaded).

Ignoring those that are tagged as *junk*, the studied repositories are listed in table 4.1. This table shows the general information found in the repository. The oldest date when the repository was initiated and the last date found in the repository at the time of this analysis. In addition, the number of commits and authors provide a first glimpse about the whole data set and its size.

[2] `http://www.mozilla.org`
[3] `http://www.mozilla.org/about/mission.html`
[4] `http://hg.mozilla.org/`

Projects	Init date	End date	Commits	Authors
Actionmonkey	2007-03-22 17:29:00	2008-07-11 18:25:38	15,926	293
Actionmonkey Tamarin	2006-11-07 05:35:54	2008-07-11 17:46:02	664	48
Camino	2002-04-20 01:53:42	2011-05-11 13:16:13	3,301	68
Chatzilla	1999-09-06 17:09:47	2011-06-01 02:39:58	1,389	89
Comm Central	2008-06-10 01:02:44	2011-06-07 13:32:06	7,899	450
Dom Inspector	2001-02-15 09:06:45	2011-06-04 15:14:27	1,169	143
Graphs	2007-08-06 02:32:33	2011-06-06 19:57:42	354	26
Ipccode	2011-02-16 12:45:58	2011-06-06 18:16:05	2	1
Mobile Browser	2008-03-18 04:05:45	2011-04-01 00:27:05	2,960	150
Mozilla Build	2006-12-14 20:44:58	2011-01-25 20:47:07	123	23
Mozilla Central	2007-03-22 17:29:00	2011-06-07 09:42:54	70,687	1,618
Penelope	2007-03-01 22:19:59	2010-10-05 19:47:13	291	10
Pyxpcom	2009-08-05 17:54:01	2011-05-02 18:00:26	60	4
Schema Validation	2004-12-10 15:11:31	2010-11-19 11:00:50	168	36
Tamarin Central	2006-11-07 05:35:54	2010-03-10 22:58:52	715	50
Tamarin Redux	2006-11-07 05:35:54	2011-06-06 18:25:07	6,380	108
Tamarin Tracing	2006-11-07 05:35:54	2008-10-02 18:17:21	536	52
Venkman	2001-04-14 19:15:10	2011-06-01 03:05:47	716	64
Xforms	2004-07-23 23:47:14	2010-12-07 22:08:36	773	40

Table 4.1: *Projects analyzed. Initial and final date. Number of commits and authors.*

In order to detail each of the analyzed projects, the following list of projects adds specific information for each project (if found). This will help to understand the role, the history of each of them and the purposes as well.

- **Actionmonkey**: the main goal of this repository was to integrate the Tamarin and SpiderMonkey projects. However it was canceled one year later because of the different speed of development between the two projects. The integration was supposed to merge SpiderMonkey and Tamarin (later detailed) into a single virtual machine that would implement the ECMAScript Edition 4 standard [5].

- **Actionmonkey Tamarin**: this repository contains the initial donation from Adobe Labs. This is the ActionScript virtual machine used in Flash 9. However, even when there were efforts to integrate both platforms, the Mozilla community is currently using their own JavaScript engine (SpiderMonkey).

- **Camino**: initially named as Chimera, Camino is a web browser based on Gecko, the Mozilla engine and designed for the Mac OS X operating systems. The main difference with the rest of the Mozilla applications is the use of the Cocoa APIs

[5]Currently, it has been released the final draft of the fifth version of this scripting language. This language is widely accepted by the industry in the form of several well known programming languages, such as JavaScript, JScript or ActionScript

instead of XUL for the user interface. The history of this repository starts in 2001 with the first release.

- **Chatzilla**: this project implements an IRC client, built on the Mozilla technology.
- **Comm Central**: this repository contains the source code of the email client, Thunderbird. Although not all of the source code to built this project is found in this repository, this contains all of the GUI development and partially part of the core system. Besides, it contains SeaMonkey, lighthing extension and Sunbird code.
- **Dom Inspector**: also known as DOMi, this is a developer tool used to edit, browse and inspect the Document Object Model of documents such as XUL windows.
- **Graphs**: created to understand the performance of several metrics in different situations: operating systems, branches and hardware.
- **Ipccode**: Container of the IPC implementations (Inter-Process Communication) mechanisms. This is use to communicate several mozilla processes to support profile sharing, among others.
- **Mobile Browser**: this repository implements the mobile browser of the Mozilla Foundation: Fennec.
- **Mozilla Build**: package that contains specific software to build Mozilla applications.
- **Mozilla Central**: current Gecko and Firefox development. Gecko is the renderingg engine used by the web browser Firefox.
- **Penelope**: this is a project based on the collaboration between QUALCOMM and the Mozilla Foundation. This project aims to develop an open source solution of Eudora Email Program.
- **Pyxpcom**: this repository contains the Python to XPCOM bridge. Thus, Python objects can talk to XPCOM or embed Gecko. This project is similar to others such as JavaXPCOM, or XPConnect (JavaScript-XPCOM bridge).
- **Schema Validation**: this is used as an extension that provides logging facilities that can help in the debugging process.
- **Tamarin Central**: this project aims to implement a high-performance, open source implementation of the ECMAScript 4th edition language specification. This code is used by Adobe as part of the ActionScript Virtual Machine within the Adobe Flash Player. This is the release repository.

- **Tamarin Redux**: this is the development repository of the Tamarin Central repository.
- **Tamarin Tracing**: this is the experimental branch of the Tamarin Central project.
- **Venkman**: this is the JavaScript debugger component.
- **Xforms**: xforms separates the data from representation in XML format.

4.1.2 General Policy

The Mozilla Foundation has developed a policy of peer review for those projects that have been defined as *core* projects and people adhering a specific project [6]. This definition identifies critical projects of the community. And those are double checked from a development point of view adding more resources to the review process. And in particular to Firefox, Thunderbird and Fennec[7].

4.1.3 Migration from CVS to Mercurial

The use of a new SCM always implies to make a decision about the source code and the possibility of migrating the whole history or starting another repository from scratch. The fastest way is to start a new repository from scratch and this is what they decided in most of the cases. However, a small set of projects decided to keep the previous history.

Figure 4.1 shows the history studied for each projects. Those projects that are older than 2,006 or 2,007 actually migrated the history, while the rest of them copied the source code from one to the other losing the previous history.

[6] `http://www.mozilla.org/hacking/committer/`. Last visited: September, 2011.

[7] `http://www.mozilla.org/hacking/commit-access-policy/` where Firefox is found in the "mozilla central" directory, Thunderbird in the "comm central" directory and Fennec in the "mobile browser" directory. All of them can be found at hg.mozilla.org. The JS Engine also has its own review policy being a "core" project: `http://www.mozilla.org/hacking/reviewers.html`. Last visited: September, 2011.

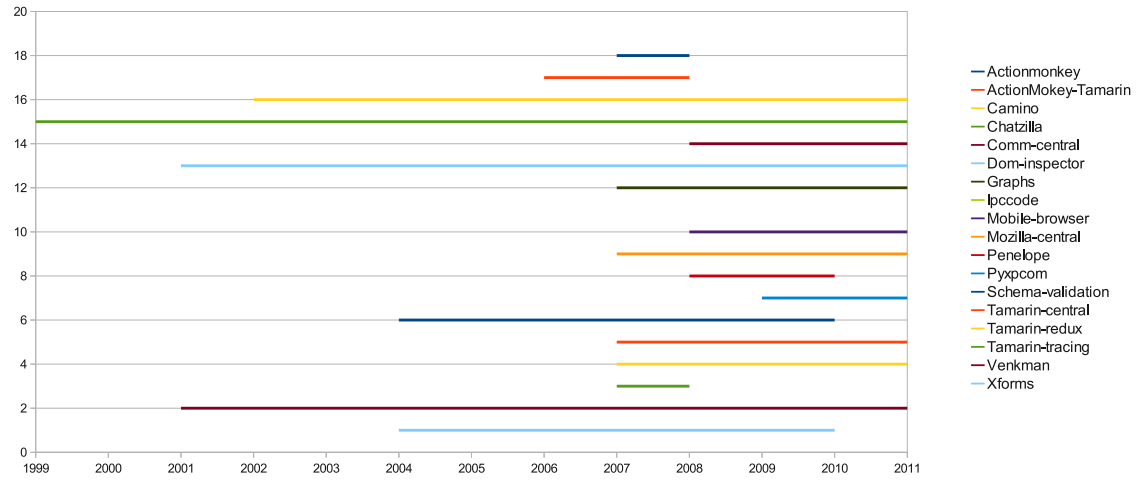

Figure 4.1: *General overview of the life of each of the projects analyzed*

4.2 Filtering the DataSet

As in any other experimental study, not all of the data is potentially useful for the researchers and there are always specific limitations to the study. This section will focus on the filters applied to the programming languages used and potential commits that should be ignored.

First, typical programming languages have been identified as being part of the Mozilla project, such as C++ o Java. Besides scripting languages such as Python or Perl are also found. And finally, the existence of other marked up languages such as XUL or HTML are massively used.

Thus, this section aims to explore the differences in terms of programming languages that are found in the cases of study and the number of source code lines and bug seeding ratio (number of bug seeding commits out of the total activity).

In summary, the main filters applied to the dataset, apart from those specified in the methodology section, and depicted in the following subsections, are the **use of traditional programming languages**, ignoring those related to marked up, such as XUL, XML or HTML and the **removal of the initial commit** from all of the calculations.

4.2.1 Programming Languages

The existence of specific programming languages in the dataset, such as XUL, being quite important to the architecture of the projects studied, may vary the results. The way those are developed, by means of using specific GUI editors, marks a difference between those and the use of IDE's or other similar and helpful tools to develop. Thus, since those are not developed in the same way, it was necessary to study how they behave. If the data present similar values in terms of activity, number of bugs, and other factors, the aggregation of the data would not present any more difficulties. However, if the marked up languages present too different results, they should probably be removed from the dataset and left them as open question for the future: the study of the way people develop depending the programming languages.

Indeed, there are differences in the way developers work depending on how verbose is a programming language (for instance Java compared to C) and even more between those born for scripting, such as Perl and Java.

In this study, only the programming languages with the highest number of lines and files are shown. The distribution per project and by programming language is detailed in tables: 4.2, 4.3, 4.4, 4.5 , 4.6, 4.7, 4.8, 4.9, 4.10, 4.11. Those are aggregated in tables 4.12 and 4.13[8].

[8]The dataset is detailed in set of two projects per table. There is not initial relationship between those two projects. This type of showing the results is done due to optimize the space and to avoid creating a table per project.

ActionMonkey			ActionMonkey Tamarin		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
C++	4,120	1,326,947	*ActionScript*	2,642	146,151
C	1,450	780,431	*C++*	190	97,826
JavaScript	3,873	423,593	*Shell*	21	26,047
C/C++ Header	4,112	406,103	*C/C++ header*	164	16,131
ActionScript	2,642	146,151	*Python*	19	2,749
Shell	224	78,938	*Perl*	1	137
Perl	259	30,817			
Python	125	14,106			
Java	162	13,325			
HTML	8,223	602,675	*XML*	18	9,426
XML	771	88,691	*HTML*	40	9,078
XUL	726	65,034			

Table 4.2: *Programming languages detected in ActionMonkey and ActionMonkey Tamarin projects*

Camino			Chatzilla		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
Shell	16	24,700	*JavaScript*	47	27,886
C	33	23,488	*Python*	6	1,320
C/C++ Header	348	14,821			
Ruby	5	575			
JavaScript	1	534			
Python	1	503			
Perl	2	231			
XML	4	754	*XUL*	17	1,319
			HTML	1	188

Table 4.3: *Programming languages detected in Camino and Chatzilla projects*

Comm Central			Dom Inspector		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
JavaScript	1,197	248,943	*JavaScript*	49	8,547
C++	387	222,891			
C/C++ Header	451	23,511			
C	26	13,966			
Python	14	1,048			
Perl	10	1,028			
XML	70	36,603	*XML*	2	762
HTML	57	1,048	*XUL*	49	3,747
XUL	425	70,824			

Table 4.4: *Programming languages detected in Comm Central and Dom Inspector projects*

Graphs			Ipccode		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
JavaScript	30	13,437	*C++*	5	3,146
Python	7	490	*JavaScript*	3	513
			C/C++ Header	4	352
HTML	2	151			

Table 4.5: *Programming languages detected in Graphs and Ipccode projects*

Mobile Browser			Mozilla Build		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
JavaScript	93	18,663	*Python*	2	261
C++	5	481	*Shell*	9	160
Shell	4	46	*C++*	1	14
XML	18	3,978			
XUL	9	1,102			

Table 4.6: *Programming languages detected in Mobile Browser and Mozilla Build projects*

Mozilla Central			Pyxpcom		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
C++	3,957	1,497,747	*C++*	28	7,627
C	1,938	998,333	*Python*	47	6,096
Javascript	7,600	727,052	*Shell*	6	2,871
C/C++ Header	5,198	589,913	*Perl*	3	735
Bourne Shell	246	110,035	*C/C++ Header*	7	685
Python	265	47,263	*Javascript*	2	114
Java	52	22,739			
Perl	182	22,100			
XUL	1,438	113,015	*XUL*	6	335
HTML	13,026	736,592	*HTML*	3	189
XML	657	52,441			

Table 4.7: *Programming languages detected in Mozilla Central and Pyxpcom project*

Schema Validation			Penelope		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
C++	12	10,786	*JavaScript*	12	4,176
C/C++ Header	7	1,407			
Javascript	3	661			
Shell	1	7			
XML	1	1,033	*XUL*	10	1,925

Table 4.8: *Programming languages detected in Schema Validation and Penelope projects*

Tamarin Central			Tamarin Redux		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
ActionScript	2,868	223,020	*ActionScript*	3,741	346,816
C++	248	124,022	*C++*	252	148,030
C/C++ Header	259	34,181	*C/C++ Header*	302	61,823
Bourne Shell	212	31,542	*Shell*	141	30,370
Python	59	13,513	*Python*	80	22,088
C	15	6,310	*C*	15	6,310
Java	20	2,596	*Java*	19	2,254
Perl	1	137	*Javascript*	86	1,325
			Perl	1	137
HTML	48	10,640	*HTML*	274	19,796
XML	21	9,716	*XML*	22	10,657

Table 4.9: *Programming languages detected in Tamarin Central and Tamarin Redux projects*

Regarding to the aggregation of the data obtaining general results, table 4.12 divides this by programming language. It is worth mentioning that the table only represents the most significative programming languages and the results are based on the tool *Cloc*, which aims to calculate the type of file and number of lines of code. This tool is similar to others such as *SLOCCount* or *Ohcount*. The number of source code lines are exactly the number of lines of code, where comments and blank lines were removed. In addition, a specific script was created to calculate the XUL files. It is observed that Javascript is the main programming language in this community, together with C/C++ and ActionScript. The existence of the last programming language is because of the Tamarin family projects. Besides, ActionMonkey and ActionMonkey Tamarin are nowadays abandoned.

In addition, table 4.13 shows a summary of the aggregated data per programming language, but also adding other variables such as commits activity, bug seeding activity and bug seeding ratio per programming language. Although these variables will be introduced and largely studied in later sections, this initial study will allow to better understand the relationship of the programming languages and these variables. This will help to filter or not the final datasets.

The results show that the type of file with less ratio of bug seeding commits out of the

Tamarin Tracing			Venkman		
Language	**Files**	**Code**	**Language**	**Files**	**Code**
ActionScript	2,647	153,526	*Javascript*	28	16,995
C++	189	89,498	*Python*	6	1,221
C/C++ Header	192	69,326	*Shell*	2	172
Shell	26	25,146			
Python	20	3,962			
C	10	1,718			
Perl	1	137			
HTML	37	7,214	*XML*	1	82
XML	12	6,263	*HTML*	3	78
			XUL	9	1,007

Table 4.10: *Programming languages detected in Tamarin Tracing and Venkman projects*

XForms		
Language	**Files**	**Code**
C++	61	20,019
C/C++ Header	29	1,772
Javascript	2	387
XML	19	9,938
XUL	5	434

Table 4.11: *Programming languages detected in XForms project*

Language	**Files**	**Code**
JavaScript	20,624	2,219,828
C/C++ Header	16,274	1,809,982
ActionScript	14,540	1,015,664
C++	13,463	5,059,977
C	5,425	2,828,889
Shell	1,176	440,500
Python	917	161,906
Perl	642	77,559
Java	305	63,653
HTML	34,749	2,124,980
XUL	4,132	371,757
XML	2,273	282,765

Table 4.12: *Aggregated data from all of the cases of study per programming language*

Language	Activity (commits)	Bug-seeding	Seeding ratio
JavaScript	30,613	22,832	74.58%
C/C++ Header	31,265	23,185	74.15%
ActionScript	1,552	986	**63.53%**
C++	52,224	37,269	71.36%
C	5,283	4,066	76.96%
Python	5,457	4,029	73.83%
Shell	4,239	3,251	76.69%
Perl	161	154	**95.65%**
Java	1,247	1,152	92.38%
HTML	11,524	8,848	76.77%
XUL	9,030	7,352	81.41%
XML	7,529	6,111	81.16%

Table 4.13: *Activity and bug seeding ratio per programming language*

activity is ActionScript. However, even when this is the fourth programming language at the level of files/lines, at the level of activity is the third one with less activity, only Java and Perl, minorities programming languages in the Mozilla community show a similar ratio of files/lines.

Another interesting point is the bug seeding ratio of the marked up languages, such as XUL or XML. They are over the typical ratio if compared to other languages such as C++ or JavaScript. Those languages, typically, are the GUI part of the Mozilla applications and they are usually developed using external tools. This usually allows developers to avoid changing the source code. This makes in some cases incomparable certain programming languages since the way they are developed is completely different.

The top languages used by the Mozilla community show a similar rate of bug seeding ratio. Thus, at the level of top programming languages, there are not main differences even when they exist from a grammatical and syntactical point of view. However, this leads to another concern related to how verbose the different programming languages are. Those files containing more lines of source code are probably more prone to have more activity that those less bigger files. This is, for instance, the case of ActionScript and C++. For a similar number of files, C++ shows around a five times the number of source code lines per files than the ActionScript programming language. This should be taken into account when measuring activity per file and bug-seeding ratio since the C++ files will probably show a higher activity. In this case study, the pretty low activity shown by the ActionScript programming language is due to this was initially imported, but later abandoned and this provokes that high level of number of source code files, but that pretty low activity in terms of commits.

Finally, a visual approach to the problem is depicted in figure 4.2. This consists of four pie charts which contain the activity measured in number of files (in the downloaded

version), the number of source code lines (in the downloaded version), the number of total commits where one or more files for a given programming language was *touched* (added, modified or removed) and, finally, from those commits, how many of them were *buggy*.

Initially, there are not main differences between C++ in the number of files and typical activity developed along the projects. Having around a 12% of the total number of files, it is around a 33% of the total commits activity. Or in other words, in a 33% of the total number of commits studied, at least one C++ file was handled.

On the other hand, we can see how the ActionScript programming language reaches around a 13% of the total files (with only a 6% of the total lines), but this is close to a 1% of the total activity. This may mean that those files based on ActionScript, were initially added to the repository, but not later modified.

With respect to the activity, only studying the C or C++ code, we are reaching around a 56% of the total activity and a similar bug seeding ratio.

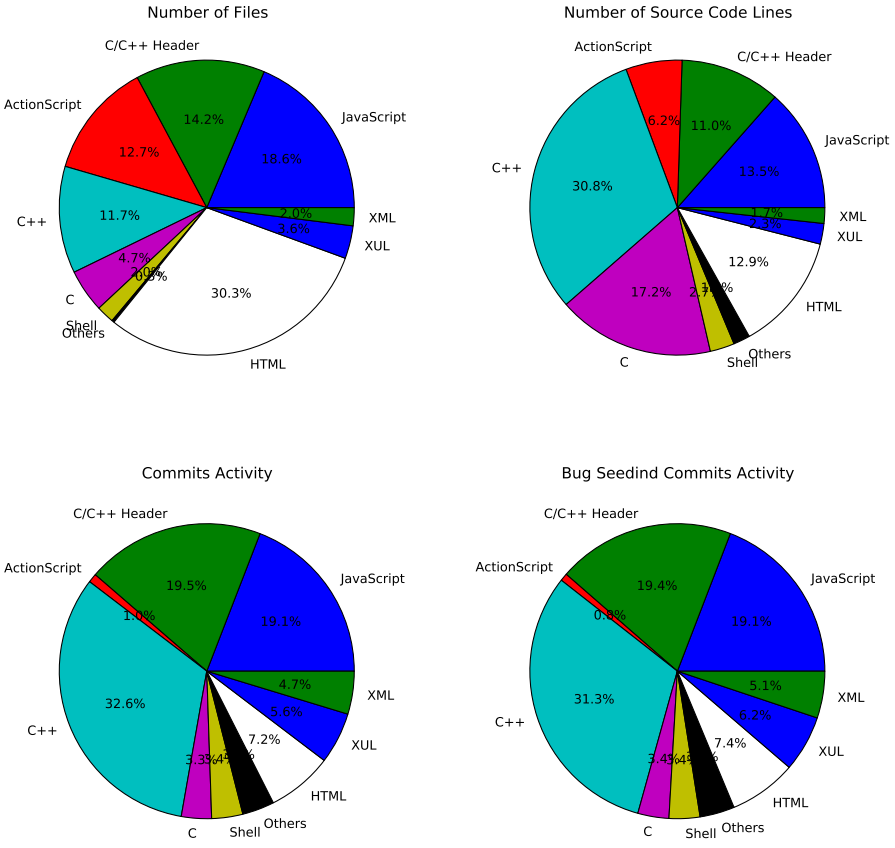

Figure 4.2: *Activity measured in files, number of source code lines, commits and bug seeding commits*

As previusly mentioned, marked up languages are mostly automatically done or written

by mean of a *WYSIWYG* (What You See Is What You Get) tools that help in the development of this type of programming languages.

Thus, along this dissertation only the following programming languages will be taken into account: **JavaScript, C/C++ Header, ActionScript, C++, C, Python, Shell, Perl and Java**, covering more than a 80% of the total number of lines.

4.2.2 Understanding large Movements of Lines

This dissertation is presenting its results at the level of commits, but basing the analysis at the granularity of lines, it is necessary to make a previous study of the different *odd* cases when studying the several SCMs.

A large movement of lines was found from some files to others. As an example when studying the comm-central repository, almost all of the lines (around 800,000) where removed in just one commit and added again after 20 commits and a couple of hours.

Since we are measuring and counting mostly commits, this does not represent a big deal for the results (those are few commits if compared to the total activity), however those lines make impossible to follow specifically the life of a line during its whole existence. As explained in the method section, this thesis is just focusing on two specific types of actions per line (modified and removed) when a commit has been detected as fixing an issue. Thus, these huge movements of lines only affect those sets of lines whose previous commit was detected as being part of these.

An example of these movements can be seen in figure 4.3 where the aggregation of the number of lines removed is studied for the Comm Central project. However, *the only commit that is filtered along this dissertation is the first commit of all.* Since most of the history has not been migrated to the new repository, the initial commit may represent hundred of thousands of lines. Thus, in the first stages of the analysis of these repositories, most of the bug fixing commits would be linked to the commit 0, where only a developer would have added the source code. Thus, in order to avoid this potential bug in the dataset, *the commit 0 was removed from the analysis.* Thus, no commits are dependant from the commit 0.

Other approaches could be used to avoid this problem, such as [Canfora *et al.*, 2007]. However, those are only applicable when files are renamed or moved in the same commit. In any case, this is left as further work and these large movements of lines should be studied together with their implications when doing research on empirical software engineering. Among others, the following causes of the big movements of lines have been detected:

- Set of lines introduced at the very beginning of the life of a project
  - Migration from another project (fork, abandoned code...)

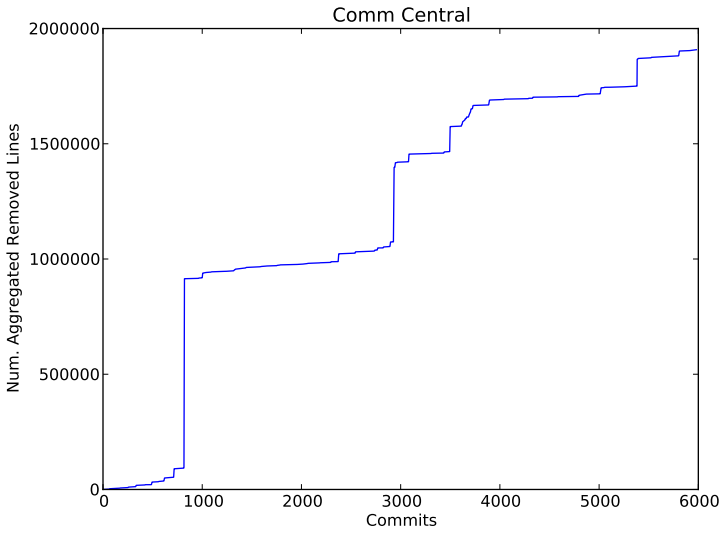

Figure 4.3: *Aggregated number of removed lines found in the Comm Central project*

  - Migration from another SCM (Mozilla community)

- Set of lines which have been removed and some commits later have been again added (comm-central)
- Set of lines which have been added and later removed
- Set of lines added (merge)
- Set of lines removed (merge)
- Set of lines modified (change of license, spaces...)
- Detection of modification of lines (not the same line, blank lines, others...)

4.3 Time to Fix a Bug

This section aims to show the time to fix a bug analysis when adding the event of bug seeding commit. As a reminder, the time to fix a bug has been previously studied with two main goals: to study the distribution of the time to fix a bug values [Gokhale & Mullen, 2010] or prediction of bugs lifetimes [Panjer, 2007], [Weiss *et al.*, 2007], [Giger *et al.*, 2010].

However the data sources used for the results are always focused on the BTS. The following set of experiments aim to study this process using the SCM as its main data source. Thus, instead of only using the event of opening and closing a bug report, this dissertation will study the point in time when a bug is *seeded* (or causing a fixing action) in the source code and the event of fixing that bug.

This new event will help to better understand the bug life cycle from the point of view of process. It will be possible to estimate the total time of fixing bugs from its very beginning. And this will help to improve the current literature when estimating, in first place, the total time to fix a bug since this is *introduced* in the source code and secondly, how much time takes to discover that bug.

Summarizing, this section is based on the bug seeding and fixing activity in the SCM and enriches the dataset adding information regarding to the opening and closing bug reports in the BTS. Thus, there are four main states, as previously declared: bug seeding commit, bug fixing commit, open bug report and close bug report.

The four events are also depicted in figure 4.4. Specifically, **T2** is the time that has been traditionally used in the literature. Thus, **T2** is the time that elapses between the event of discovering a bug in the source code (opening the correspondent bug report) and closing it (fixing it in the source code and closing the bug report). Regarding to **T1** this is the time between the moment when a bug was seeded in the source code and the event when this is discovered. Thus, **T1 + T2** is the time between the event of involuntary introducing a bug in the source code and fixing that bug in the source code.

Regarding to the set of experiments, those have been divided in four different studies:

1. **Time elapsed between bug seeding and bug fixing commit (T1 + T2)** (section 4.3.1): this is the time of fixing a bug only taking into account information provided by the SCM.

2. **Time elapsed between the open bug and close bug report (T2)**(section 4.3.2): this is the time of fixing a bug only considering information from the BTS.

3. **Time elapsed between open bug report and bug seeding commit (T1)** (section 4.3.3): this is the time between the action of involuntary seeding an issue

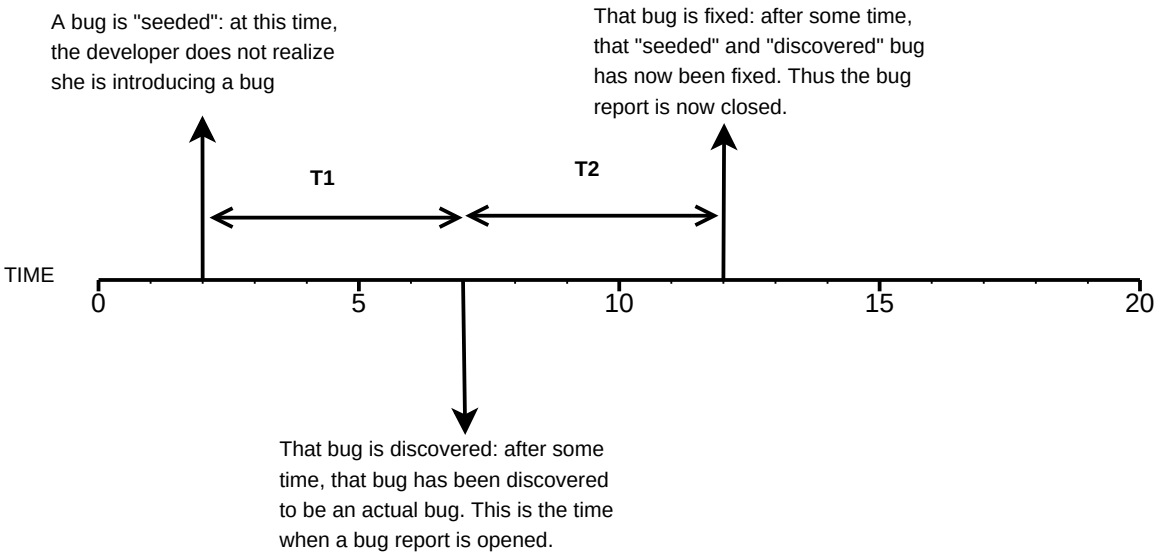

Figure 4.4: *Bug life cycle diagram: time to fix a bug*

in the source code and discovering it by a developer (opening the correspondent bug report).

4. **Time elapsed between close bug report and bug fixing commit (end of T2)** (section 4.3.4): this is the time of fixing a bug in the SCM and closing the correspondent bug report. It is expected to have similar values.

In addition, the distribution of the time-to-fix-a-bug is studied for each project and data source (subsection 4.3.5 and a final discussion subsection is added (subsection 4.3.6).

It is worth stressing the point that all of the times of the SCM have been translated to a neutral timezone, (+00:00 was selected for this purpose) in order to actually compare datasets with the same characteristics. With respect to the BTS dataset, the times are always stored in "PDT" timezone and were equally translated to the (GMT +00:00) timezone in order to be comparable to the dataset used from the SCM.

4.3.1 Time to Fix a Bug: Bug Seeding and Bug Fixing Commits

This subsection presents the results in time between the bug fixing commits and its relatives bug seeding commits in time. For this purpose, the time when the commit was submitted has been translated to GTM +00:00 in order to make them comparable. As explained, the definition of a bug seeding commit is a commit whose changes, or part of them, have been later detected as being involved in a bug fixing commit. Thus, depending on the previous number of commits, there may appear specific situations when calculating this time to fix a bug:

- **Type 1**: Each pair of bug seeding and bug fixing commit is separately calculated from the rest. Thus, if a bug fixing commit was found to be seeded in four previous commits, those differences in time are taken one-one.

- **Type 2**: In this case, for each set of bug seeding commits obtained and dependent of a bug fixing commit, the time is calculated as the average of all of them.

- **Type 3**: In this case, for a given set of bug seeding commits, only the earliest of them is used, the rest are ignored.

- **Type 4**: In this case, for a given set of bug seeding commits, only the latest is used to calculate the difference, the rest are ignored.

Table 4.14 and 4.15 show an early statistical approach to the average time to fix a bug seeding commit. The projects *Ipccode*, *Mozilla Build*, *Pyxpcom* and *Schema Validation* presented no data and the results for each of the cases is 0. Thus, they were removed from the table.

In addition, table 4.16 presents the results of the errors found in the dataset. In few cases the time when a commit fixing an error happened before the event of *seeding* that commit. However, by definition, the time of seeding errors in the source code must happen in all of the cases before the time when those errors are fixed. In fact, the commits are sequentially found, however the times specified in the SCM are wrong. For this purpose, part of the data was filtered to avoid these *odd* cases. Besides, the percentage of values removed is specified.

As seen in the data and the type of analysis used (the earliest, the latest and so on), there are significant differences in terms of mean and median. Those results using the type 1 (each bug seeding-fixing commit is a value) presents a higher set of values in terms of mean and median, but also in the standard deviation. The main reason for this distinction in the final values is the previous number of seeding commits linked to a fixing commit. It has been observed (section 4.5) that in around a 30% or 40% of the cases, depending on the project, a bug fixing commit is linked to at least 2 or more previous bug seeding commits.

Thus, those differences in terms of the final results are partially explained by these characteristics of the dataset. The higher the percentage of bug seeding commits linked to a bug fixing commit, the higher the differences. Thus, this peculiarity provokes an increase in parts of the results.

From the perspective of the committer that is seeding bugs, the bug seeding commit could be seen as the basic piece of information (instead of lines). If a committer has seeded a bug, it is not significant (at least for the very definition of bug seeding commit

Project	type	max	min	mean	median	std
Actionmonkey	1	468.89	0.00	75.12	35.73	90.77
	2	445.20	0.00	62.93	34.21	74.78
	3	445.20	0.00	61.07	20.62	84.73
	4	466.42	0.00	64.15	25.22	85.19
Actionmonkey Tamarin	1	602.86	0.01	81.12	19.03	123.45
	2	445.26	0.01	70.06	22.13	94.68
	3	470.08	0.01	79.43	8.30	125.91
	4	485.77	0.01	65.54	13.77	111.94
Camino	1	3220.90	0.00	1185.48	1103.66	870.74
	2	3061.45	0.51	823.15	688.24	710.55
	3	3155.49	0.51	842.95	666.55	771.29
	4	3179.63	0.51	831.15	530.28	819.05
Chatzilla	1	4097.89	0.00	897.43	717.97	777.25
	2	3138.15	0.00	748.26	586.94	657.83
	3	3618.81	0.00	785.26	553.94	768.45
	4	4097.89	0.00	751.41	497.05	748.34
Comm Central	1	944.62	0.00	229.61	116.04	245.61
	2	944.62	0.00	207.31	135.92	211.63
	3	944.80	0.00	225.15	119.82	248.87
	4	944.62	0.00	198.10	90.80	230.74
Dom Inspector	1	3608.53	0.00	1004.56	809.75	1005.58
	2	3519.76	0.00	789.38	592.60	737.02
	3	3519.76	0.00	786.64	389.63	950.94
	4	3608.53	0.00	765.79	431.59	882.49
Graphs	1	1134.83	0.00	200.92	116.21	210.65
	2	1134.83	0.31	200.16	130.52	202.82
	3	1134.83	0.31	193.94	101.27	231.70
	4	1134.83	0.00	207.64	128.70	229.72
Mobile Browser	1	951.56	0.00	141.43	77.94	164.72
	2	874.99	0.00	104.29	66.78	114.25
	3	874.99	0.00	107.23	50.09	141.02
	4	874.99	0.00	95.27	45.48	128.52
Mozilla Central	1	1183.10	0.00	115.24	24.73	198.28
	2	1167.91	0.00	80.02	25.37	128.50
	3	1177.32	0.00	76.59	15.85	144.79
	4	1167.91	0.00	80.24	15.90	155.06

Table 4.14: *Early Statistical Approach (in days) (Part I): time to fix a bug in the source code*

Project	type	max	min	mean	median	std
Penelope	1	1156.48	0.00	311.13	218.10	319.71
	2	1119.00	0.02	353.42	296.37	275.90
	3	1119.00	0.01	400.16	343.01	351.04
	4	1119.00	0.02	328.22	118.42	338.97
Tamarin Central	1	678.78	0.01	123.85	56.82	154.20
	2	445.26	0.01	102.84	62.00	110.28
	3	509.97	0.01	103.21	23.40	141.95
	4	445.26	0.01	84.87	26.24	118.45
Tamarin Redux	1	1610.09	0.00	197.82	117.21	230.36
	2	1049.03	0.00	143.96	107.80	152.12
	3	1610.09	0.00	134.55	58.31	186.94
	4	1309.74	0.00	144.57	67.32	196.71
Tamarin Tracing	1	541.94	0.01	94.16	53.83	113.90
	2	369.02	0.01	79.98	53.04	82.71
	3	376.78	0.01	80.96	32.92	102.80
	4	400.84	0.01	79.10	33.82	106.57
Venkman	1	3536.71	0.01	798.51	361.22	951.96
	2	3228.08	0.00	995.90	860.49	851.63
	3	3357.08	0.00	993.66	764.58	950.32
	4	3228.08	0.00	938.99	709.44	912.44
Xforms	1	2141.78	0.02	577.53	397.02	552.39
	2	2047.29	0.02	436.96	235.73	514.20
	3	2141.78	0.02	454.30	242.25	565.48
	4	2070.99	0.01	403.28	163.20	515.77

Table 4.15: *Early Statistical Approach (in days) (Part II): time to fix a bug in the source code*

Project	Total	Filtered
Actionmonkey	11,113	253 (2.27%)
Actionmonkey Tamarin	307	3 (0.97%)
Camino	366	0 (0%)
Chatzilla	1,940	4 (0.2%)
Comm Central	11,970	97 (0.81%)
Dom Inspector	868	8 (0.92%)
Graphs	208	0 (0%)
Mobile Browser	7,213	85 (1.17%)
Mozilla Central	134,115	7,532 (5.61%)
Penelope	86	0 (0%)
Tamarin Central	305	22 (7.21%)
Tamarin Redux	8,874	140 (1.57%)
Tamarin Tracing	441	22 (4.98%)
Venkman	549	0 (0%)
Xforms	424	0 (0%)
Total	178,779	8,166 (4.56%)

Table 4.16: *Population used and filtered. Data measured in number of couples of bug seeding and bug-fixing commits detected in the SCM per project. The filtered values are those couples whose time of fixing commit happened before the time of seeding commit in the SCM.*

) if there are others in the past or future also seeding that bug (growing that bug with other changes). The fact is that a bug was seeded in the source code.

With this respect, figure 4.5 shows the density chart associated to the time to fix a bug since this has been *seeded*. The values represented are the results of type 1 where each couple bug seeding - bug fixing commit is represented. The values are calculated in log10 scale to flatten the charts. Peaks of density are all found between the values 1.0 (10 days) to 3.0 (1,000 days) to fix a bug.

Either in figure 4.5 or tables 4.14 and 4.15 there is a big difference between a first set of projects, where usually tend to be between 100 and 200 days to fix a bug in average (graphically observed in the charts). And those that show a higher rate (not really seen in the density charts since they are *log10* based calculation, but seen in the table results, where they reach around 400 days in average).

Thus, a new question is raised based on these results and this is related to the difference in terms of days. The first and quick answer is based on the history analyzed in the dataset. The older the history of a project, the more likelihood of finding older bugs. If the results are checked, and compared to table 4.1 where the total number of months analyzed per project is depicted, the extremes are *Actionmonkey* plus *Actionmonkey Tamarin*, where the life is around 1 year. However, although those projects are the ones that fix a bug in the fastest time, others show closer times to fix a bug such as *Mozilla Central*.

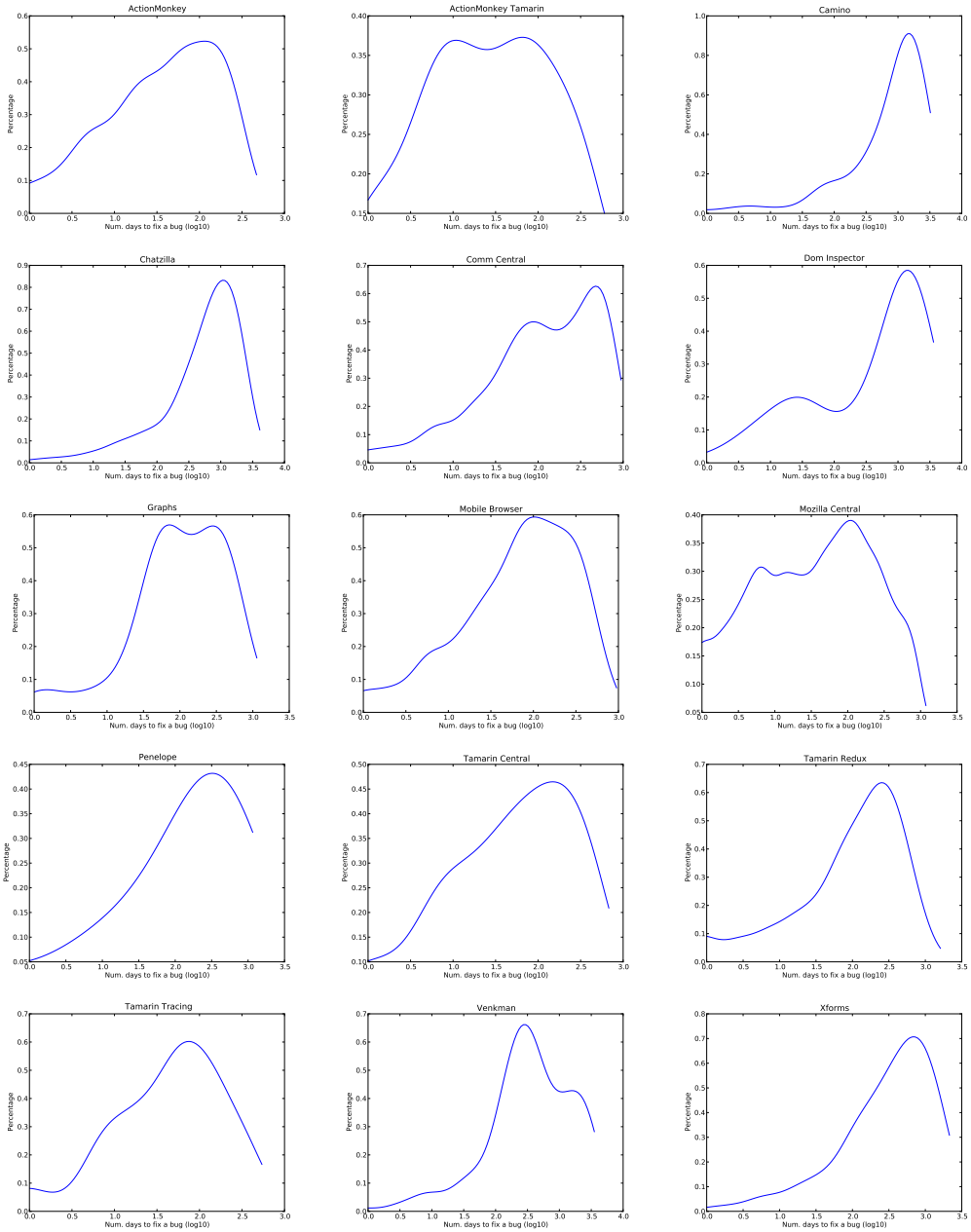

Figure 4.5: *Time to fix an issue: density charts calculated in log10.*

4.3.2 Time to Fix a Bug: Open Report - Close Report in BTS

This subsection aims to carry out a similar analysis, but focused on the bug tracking systems as the data source where the information is retrieved. As seen in previous literature (chapter 2, section 2.2), the BTS is the general type of data source used when the time to fix a bug is calculated.

Thus, this subsection will study the time to fix a bug from those bugs that are specifically detailed in the *log message* left by the developers in the SCM. Based on that

Project	Retrieved bugs	Studied bugs	Strange dates
ActionMonkey	6,766	3,205	80 (2.49%)
ActionMonkey Tamarin	111	75	4 (5.33%)
Camino	640	113	2 (1.76%)
Chatzilla	630	456	14 (3.07%)
Comm central	4,312	2,186	63 (2.88%)
Dom Inspector	377	156	3 (1.92%)
Graphs	123	42	1 (2.38%)
Ipccode	0	0	0 (0%)
Mobile browser	2,132	1,302	105 (8.06%)
Mozilla build	72	0	0 (0%)
Mozilla central	31,213	19,795	481 (2.42%)
Penelope	59	25	0 (0%)
Pyxpcom	0	0	0 (0%)
Schema validation	61	0	0 (0%)
Tamarin central	116	66	4 (6.06%)
Tamarin redux	1,266	961	20 (2.08%)
Tramarin tracing	76	47	3 (6.38%)
Venkman	150	95	2 (2.10%)
Xforms	155	88	1 (1.13%)
Total	*48,259*	*28,612*	*783 (2.73%)*

Table 4.17: *Retrieved report issues from Bugzilla of Mozilla based on the information found at the log message. In addition, a set of them has been filtered due to errors in the dates of the BTS between the time when a bug report was open and closed. Specifically those where the close report time happened earlier than the even of opening the bug report.*

log message, a *bug id* is obtained. Table 4.17 shows the number of bug reports retrieved from the Bugzilla of Mozilla (first column). This initial set of issues are retrieved from the Bugzilla. Those are later filtered taking only those that matched with the source code management system (second column).

Finally, it has been observed, as in the case of the SCM, that in specific cases the time when a bug declared to be closed happened before the event of opening its bug report. This seems again to be oddities in the dataset that are removed.

Based on the 28,612 bugs retrieved from the Bugzilla of Mozilla, table 4.18 shows an early statistical approach. The data is presented in days to fix a bug in the BTS.

Finally, figure 4.6 depicts the density charts of the time elapsed between the event of opening a bug report and closing it. The values are calculated in *log10* scale to flatten the charts. Peaks of density are all found between the values 0 (1 days) to 2.0 (100 days) to fix a bug. However, if compared to the previous time to fix a bug in the source code, there is a clear decrease where it was usual to find peaks of density close to 3.0 (1,000) days.

As it is observed, specific projects lead again the list of projects that are the fastest when fixing errors in the source code.

Project	max	min	mean	median	std
Actionmonkey	3320.92	0.0	141.35	11.91	385.83
Actionmonkey Tamarin	1824.81	0.0	136.24	16.04	274.37
Camino	2869.86	0.00	171.00	10.16	425.41
Chatzilla	2833.75	0.00	275.50	51.98	529.42
Comm Central	4001.13	0.00	191.98	12.01	509.92
Dom Inspector	3285.23	0.00	468.80	56.91	879.54
Graphs	569.10	0.81	93.58	106.67	73.00
Mobile Browser	817.86	0.00	30.14	4.66	75.78
Mozilla Central	4334.07	0.00	114.29	16.05	343.70
Penelope	1105.06	0.77	287.77	32.07	428.40
Tamarin Central	1824.81	0.00	121.23	11.09	271.78
Tamarin Redux	1824.81	0.00	70.25	18.83	131.91
Tamarin Tracing	1824.81	0.04	107.14	35.00	253.73
Venkman	2833.75	0.00	305.28	14.09	570.88
Xforms	2833.75	0.05	186.22	65.00	401.91

Table 4.18: *Early Statistical Approach (in days): Open Bug Report - Close Bug Report*

4.3.3 Open Bug Report and Bug Seeding Commit

So far it has been studied the time to fix a bug from two different perspectives: in the source code management system (in figure 4.4: **T1 + T2**) and in the bug tracking system (in figure 4.4: **T2**). This subsection aims to focus on the differences between the time to seed a bug and the time when this is discovered. In figure 4.4 this is represented as **T1**.

It is expected to find a seeded bug before the bug report is open. Or in other words, the bugs should exist in the source code before it has been detected. However, the results show a different scenario when using the methodology proposed in this dissertation. In some cases, some bug reports are opened in the bug tracking system before it has been detected as seeded in the source code.

According to table 4.19, the results show how in specific projects, such as the mozilla-central repository (the Firefox project) there is even a 45% of the bug fixing commits that are affected by this issue[9].

A plausible explanation for this behaviour is that the current approach is only aware of the last change made to those set of lines that were *modified* or *removed* in the bug fixing commit. Thus, the error may have been previously seeded, but the methodology only retrieves the previous change to the bug fixing commit. In addition to that limitation, several bug seeding commits may be part of a later bug fixing commit.

As initial results, it is interesting to appreciate the differences between those projects

[9]With respect to the dataset itself, table 4.19 focuses on bug fixing commits and table 4.18 focuses on bugs retrieved from the BTS. As expected, the total values are different. This indicates that for each bug, there may appear more than one bug fixing commit.

Project	BSC earlier	Earlier bug reports	Total fixing commits
Actionmonkey	2,081	1,466 (41.33%)	3,547
Actionmonkey Tamarin	62	23 (27.05%)	85
Camino	102	15 (12.82%)	117
Chatzilla	455	72 (13.66%)	527
Comm Central	1,788	648 (26.60%)	2,436
Dom Inspector	156	27 (14.75%)	183
Graphs	41	4 (8.88%)	45
Ipccode	0	0 (0%)	0
Mobile Browser	1,227	232 (15.90%)	1,459
Mozilla Build	0	0 (0%)	0
Mozilla Central	13,827	11,173 (44.69%)	25,000
Penelope	22	3 (12.0%)	25
Pyxpcom	0	0 (0%)	0
Schema Validation	0	0 (0%)	0
Tamarin Central	60	14 (18.91%)	74
Tamarin Redux	1,007	347 (25.62%)	1,354
Tamarin Tracing	47	10 (17.541%)	57
Venkman	97	6 (5.82%)	103
Xforms	79	14 (15.05%)	93
Total	21.051	14,054 (40.03%)	35,105

Table 4.19: *Number of open bug report whose opening date happened before the bug seeding commit date*

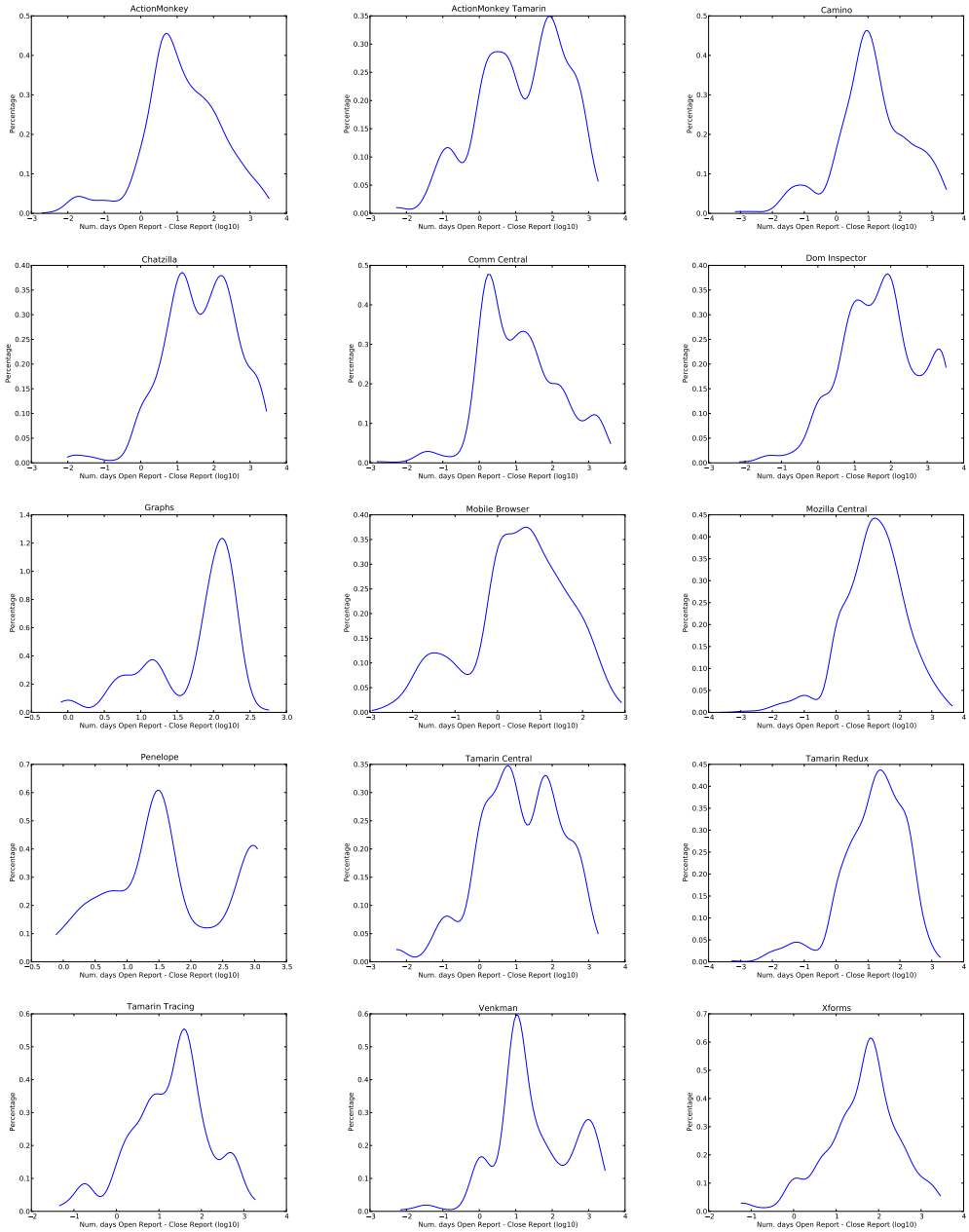

Figure 4.6: *Density charts showing the differences between the time when a bug report was opened and closed in the BTS*

named as *core* by the community and the rest of them. In that first set of projects, the percentage of reports opened before that the bug seeding is much higher if compared to the rest of the projects (although the number of total analyzed commits is not that high for the non-core projects).

The highest value is reached by the Mozilla Central project where a 45.22% of the total number of open bug reports is found to be earlier if compared to the respective bug seeding commit. This may raise another question related to the limitation of this study, however,

Project	max	min	mean	median	std
Actionmonkey	463.95	-3287.97	-68.78	7.55	403.47
Actionmonkey Tamarin	602.32	-1018.22	44.11	5.78	183.97
Camino	3213.49	-2548.22	989.77	943.36	944.69
Chatzilla	3942.23	-2210.48	700.06	586.25	860.48
Comm Central	941.94	-3975.02	36.86	75.84	598.98
Dom Inspector	3550.59	-3283.49	596.59	387.45	1253.29
Graphs	1134.41	-423.37	115.96	61.68	239.82
Mobile Browser	951.31	-3636.52	112.26	57.68	189.76
Mozilla Central	1171.20	-4308.34	-6.47	1.09	399.02
Penelope	1123.57	-1105.93	19.21	79.70	543.13
Tamarin Central	668.35	-1018.22	94.68	37.13	209.25
Tamarin Redux	1609.60	-3177.52	129.46	72.87	271.94
Tamarin Tracing	475.87	-1018.22	68.09	34.06	167.05
Venkman	3347.68	-2581.20	587.06	303.34	925.53
Xforms	2140.73	-1399.76	456.27	284.42	584.47

Table 4.20: *Early Statistical Approach (in days): Open Bug Report - Bug Seeding Commit*

what it is clear is that this dataset is providing a finer granularity when understanding a distributed environment such as the Mozilla Foundation.

Figure 4.7 shows the differences in the density charts among the several projects. The long left tail found in some of the density charts suggest that for a subset of each of the couples bug seeding, there may appear cases where the bug was earlier detected than the bug seeding commit. As a result, it may indicate that the approach used is in some cases not approximating effectively the results, since some bug seeding commits could appear before. These results are also found in table 4.20.

Improving the Method

It has been observed how using this specific methodology, the use of the diff tool is not as good as expected. Using the assumption that the previous changes of a fixing commit are the ones responsible of causing that fixing action, it has been demonstrated how in some cases (reaching a 45% of the total fixing commits analyzed), this assumption is not valid.

Thus, this subsection aims to study this process in depth only comparing those bug seeding commits and open reports that fit in that assumption. This will help to understand the process of fixing a bug and the potential time needed as seen in figure 4.4.

This experiment requires to remove those commits where any of its bug seeding commits events happened later than the open bug report.

This experiment will help to understand and compare the time to fix a bug from its very beginning to its very end. In order to work with the correct dataset and avoid methodological issues, this experiment is built on the following limitations:

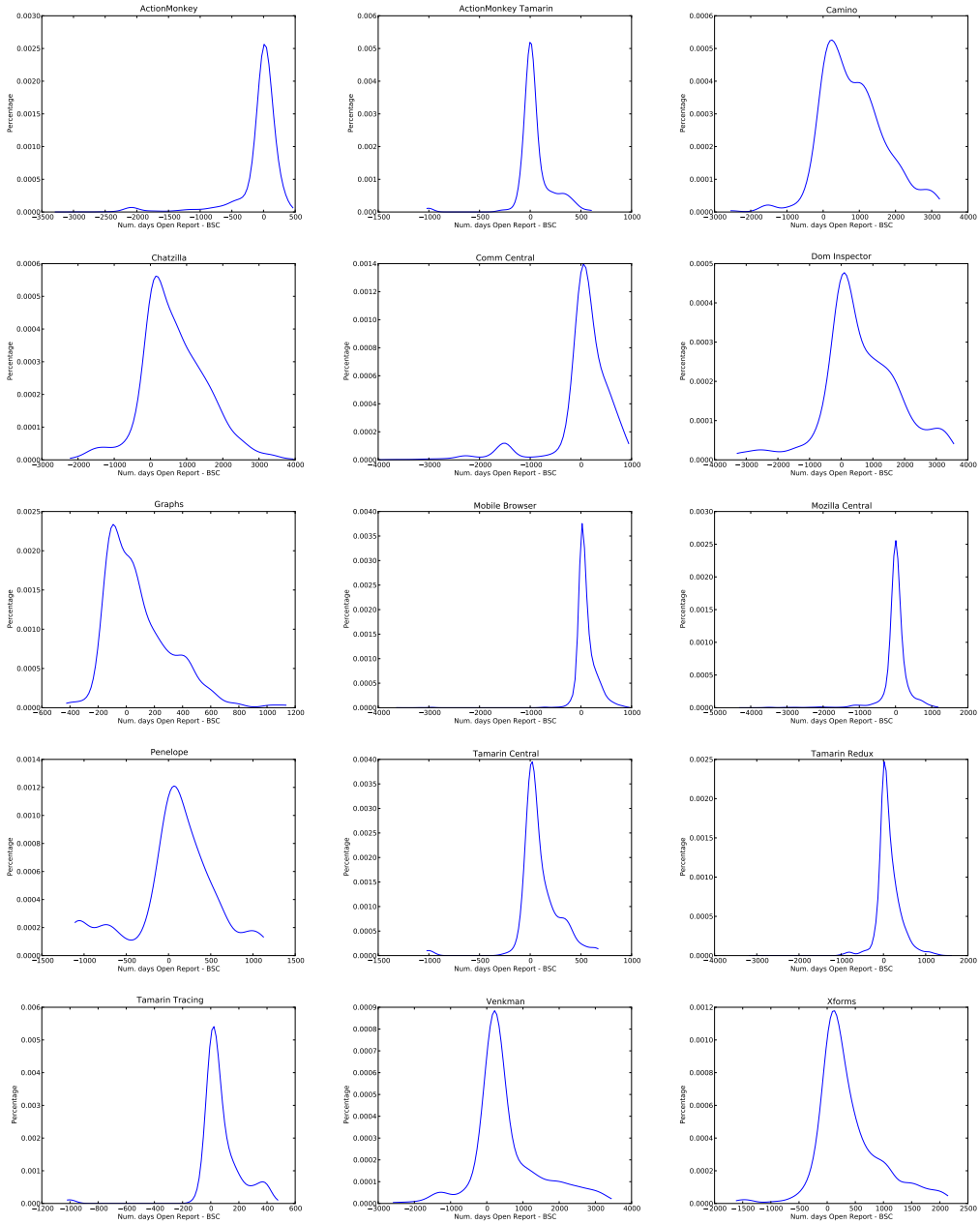

Figure 4.7: *Density charts showing the differences between the time when a bug report was opened and the bug seeding commit*

1. The fixing commits must have changed only source code (general limitation of the whole thesis).

2. The fixing commits must be linked to an entry in the bug tracking system through the bug id specified in the commit log.

3. The time of the bug open report must be younger than at least one of the bug seeding commits associated to the bug fixing commit.

Project	max	min	mean	median	std
ActionMonkey	463.96	0.02	111.47	84.09	100.98
ActionMonkey Tamarin	602.33	0.73	166.42	124.44	168.77
Camino	3213.49	0.57	1241.65	1062.27	938.13
Chatzilla	3942.23	0.61	1057.98	808.76	886.19
Comm Central	941.94	0.00	294.00	218.94	261.51
Dom Inspector	3550.59	0.05	1300.23	946.88	1133.13
Graphs	1134.41	51.47	332.75	256.15	258.36
Mobile Browser	951.31	0.10	204.91	132.68	205.92
Mozilla Central	1171.21	0.01	189.27	82.89	240.52
Penelope	1123.58	7.64	497.65	457.96	326.91
Tamarin Central	668.36	0.73	203.23	174.56	193.00
Tamarin Redux	1609.61	0.08	314.33	240.03	292.51
Tamarin Tracing	475.87	0.73	155.61	130.64	144.91
Venkman	3347.69	1.40	1221.27	989.51	971.29
Xforms	2140.74	2.08	571.52	387.44	589.43

Table 4.21: *Early Statistical Approach (in days): Bug Seeding Commit - Open Bug Report. Improved analysis.*

4. The time to calculate the time to fix a bug, based on figure 4.4 is the time of the oldest bug seeding commit minus the time when the bug was discovered and reported.

Thus, now only taking into account the *mean* when estimating the time to fix a bug, table 4.23 shows the value of **T1** and **T2**. As it can be seen, the time between the first seeding action until this is discovered by the community represents between a 60% and a 90% of the total life of that bug. In other words: since it is pretty complicated to detect bugs in advance, those tend to live in the source code for a long time. On the other hand, the time to fix that bug since this has been reported in the bug tracking system represents between a 40% and a 10% of the total life of that bug.

Based on these results, it is interesting the study of potential causes of seeding issues in the source code. Knowing in advance that the time between the bug is seeded in the source code until this is discovered is so high that should force communities to improve the detection of seeding changes. Indeed, this is part of the goals of this thesis: once it is better understood how much time takes to fix a bug, it is mandatory to study the potential causes of bug introduction changes.

This bug life cycle is also depicted in figure 4.8 where the periods in other color represents the 60% - 90% of the time that a bug needs to reach to be discovered in average.

Project	max	min	mean	median	std
ActionMonkey	1019.27	0.00	27.45	7.85	67.78
ActionMonkey Tamarin	602.90	0.01	104.26	7.96	187.78
Camino	1824.82	0.00	143.35	14.07	349.16
Chatzilla	2833.75	0.01	179.60	47.07	372.06
Comm Central	804.96	0.00	45.52	10.12	95.99
Dom Inspector	3285.23	0.01	328.06	33.32	692.56
Graphs	248.00	0.81	59.93	63.02	56.30
Mobile Browser	473.22	0.00	17.97	4.10	41.58
Mozilla Central	1039.82	0.00	29.47	7.94	68.37
Penelope	743.05	0.78	93.24	14.58	186.36
Tamarin Central	602.90	0.01	96.95	8.87	183.66
Tamarin Redux	617.88	0.00	47.72	13.83	85.70
Tamarin Tracing	602.90	0.05	106.19	11.95	196.67
Venkman	2581.28	0.01	206.35	22.10	425.48
Xforms	992.98	0.06	98.17	36.59	164.98

Table 4.22: *Early Statistical Approach (in days): Open Bug Report - Close Bug Report*

Project	mean T1	mean T2	Total time (days)
ActionMonkey	111.47 (80.24%)	27.45	138.92
ActionMonkey Tamarin	166.42 (61.48%)	104.26	270.68
Camino	1241.65 (89.64%)	143.35	1,385
Chatzilla	1057.98 (85.49%)	179.60	1,237.52
Comm Central	294.00 (86.59%)	45.52	339.52
Dom Inspector	1300.23 (79.85%)	328.06	1,628.29
Graphs	332.75 (84.73%)	59.93	392.68
Mobile Browser	204.91 (**91.89%**)	17.97	222.98
Mozilla Central	189.27 (86.52%)	29.47	218.74
Penelope	497.65 (84.22%)	93.24	590.89
Tamarin Central	203.23 (67.70%)	96.95	300.18
Tamarin Redux	314.33 (86.81%)	47.72	362.05
Tamarin Tracing	155.61 (**59.43%**)	106.19	261.8
Venkman	1221.27 (85.54%)	206.35	1,427.62
Xforms	571.52 (85.34%)	98.17	669.69

Table 4.23: *Bug life cycle: time to discover a bug and time to fix a bug*

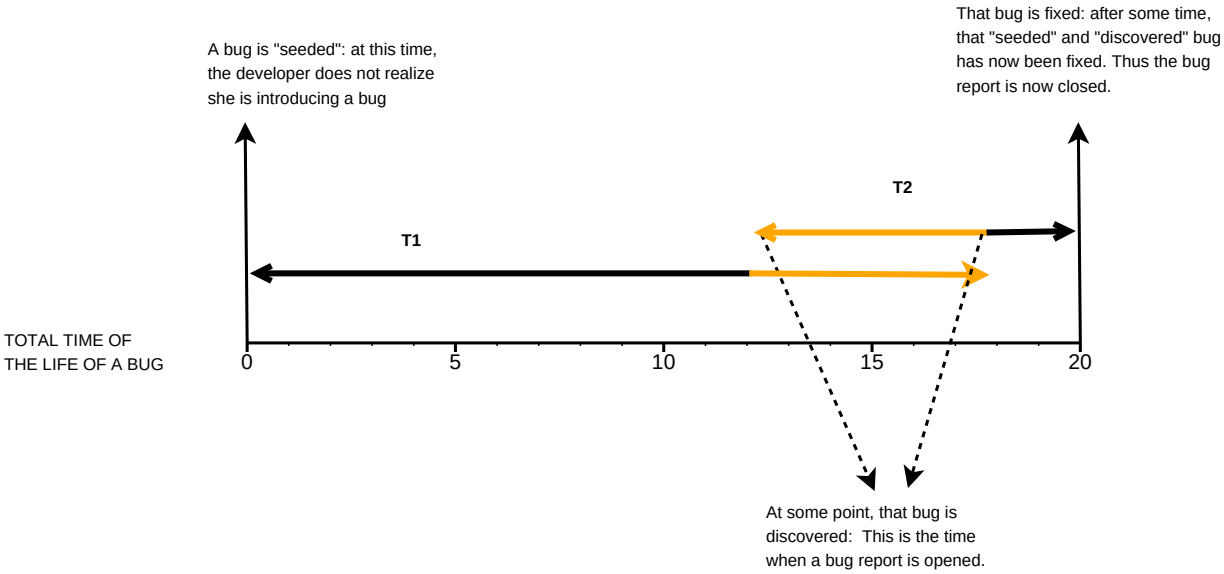

Figure 4.8: *Time to fix a bug diagram*

4.3.4 Close Bug Report and Bug-fixing Commit

This subsection aims to measure the differences in time when closing a bug report in the BTS and fixing that bug in the source code. In this case, the results show the number of days that, for a given bug fixing commit, the closing report was done. Table 4.24 shows a basic statistical approach of the data presented.

In addition, figure 4.9 depicts the density charts.

It is observed than in most of the cases there is a huge peak close to 0. This means that in most of the cases, the time to fix a bug in the source code and the time when a bug report is closed happen pretty close in time. In any case, long tails have been detected in both ways: a bug was fixed in the source code and long time later closed its bug report. Or that a bug report was closed in the BTS but it was not fixed until long time later.

4.3.5 Distributions of the Data

It has been studied that the distribution of fixing activities usually follow a Laplace Transform of the Lognormal distribution [Gokhale & Mullen, 2010]. However, it has not been studied if the time to fix a bug in the source code follows the same distribution as the time to fix a bug in the BTS. In addition a study of the normality or log-normality of the distributions is added. Although the BTS information has previously been observed to follow another type of distribution, this information is also added for completeness reasons. In order to improve the understanding of the distributions, a set of null hypotheses is detailed to validate them using an statistical approach. A summary of the several null hypotheses is developed in table 4.25.

Project	max	min	mean	median	std
Actionmonkey	1013.64	-1982.06	-4.49	-0.41	94.701
Actionmonkey Tamarin	749.54	-88.24	90.23	-0.04	192.81
Camino	1490.32	-40.48	11.09	-0.44	121.72
Chatzilla	2317.26	-317.37	74.70	-0.41	359.25
Comm Central	803.52	-2660.97	-0.30	-0.03	73.78
Dom Inspector	2645.07	-1352.12	39.04	-0.41	281.60
Graphs	50.48	-8.43	2.89	-0.41	10.15
Mobile Browser	480.72	-3606.46	0.67	-0.41	58.41
Mozilla Central	1013.64	-3606.46	-1.43	-0.14	79.18
Penelope	6.21	-69.63	-4.87	-0.83	14.01
Tamarin Central	749.54	-88.24	81.82	0.00	181.22
Tamarin Redux	749.54	-3173.25	0.58	0.05	112.03
Tamarin Tracing	749.54	-20.43	72.79	0.00	170.32
Venkman	1780.36	-223.32	31.06	-0.01	169.40
Xforms	1344.17	-583.76	57.72	-0.03	225.42

Table 4.24: *Early Statistical Approach (in days): Close Bug Report - Bug Fixing Commit*

ID	Null hypothesis	Test applied	Attribute
$H_{0,0}$	Time to fix a bug obtained from the source code and from the bug tracking system follow the same distribution	Kolmogorov-Smirnov	Time to fix a bug in SCM and BTS
$H_{0,1}$	Time to fix a bug obtained from the source code follows a normal distribution	K-S against a normal dist.	Time to fix a bug in SCM
$H_{0,2}$	Time to fix a bug obtained from the source code follows a log-normal distribution	(same)	Time to fix a bug in SCM
$H_{0,3}$	Time to fix a bug obtained from the BTS follows a normal distribution	(same)	Time to fix a bug in BTS
$H_{0,4}$	Time to fix a bug obtained from the BTS follows a log-normal distribution	(same)	Time to fix a bug in BTS

Table 4.25: *Null Hypotheses on the distributions of the time to fix a bug in the SCM and BTS*

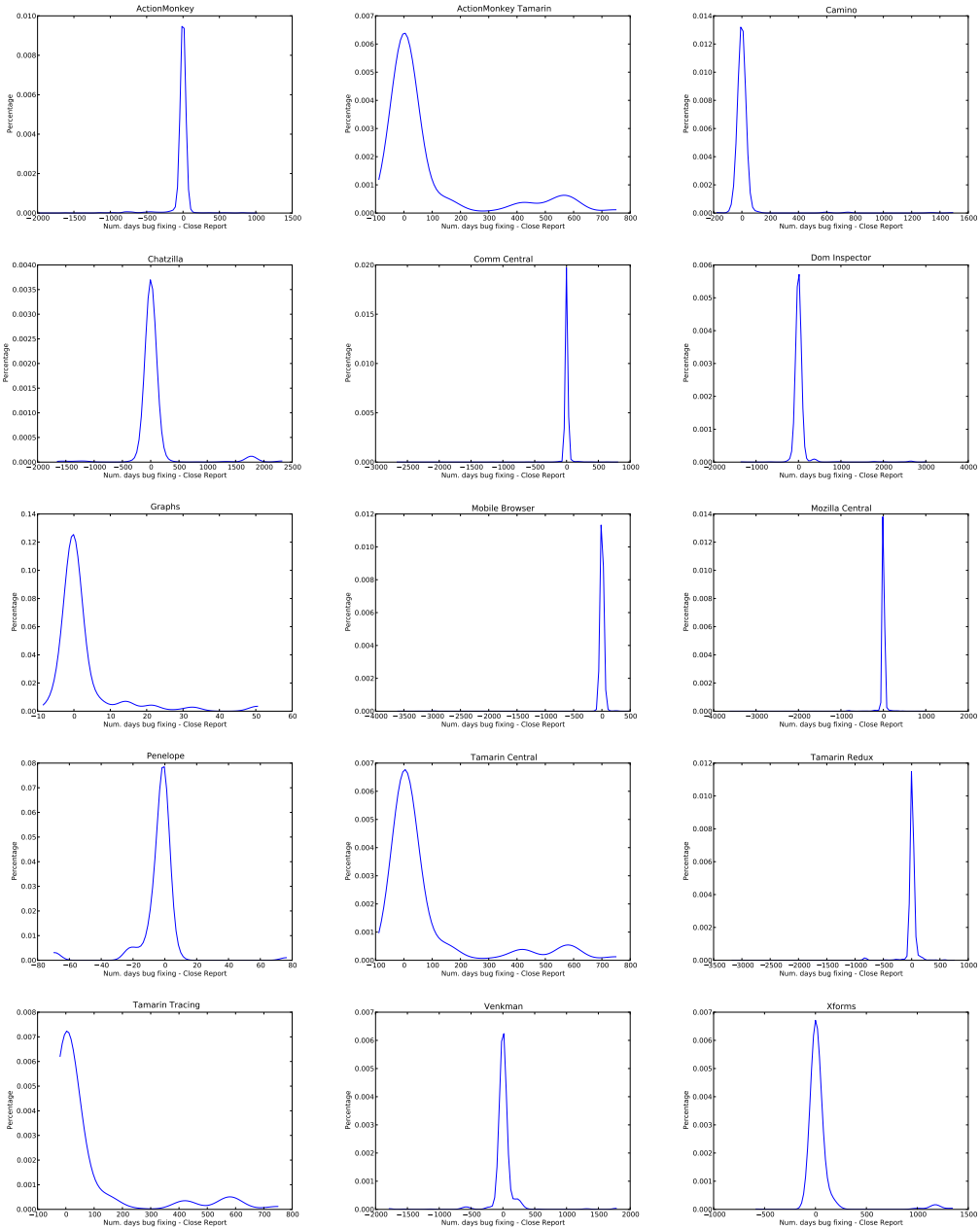

Figure 4.9: *Density charts showing the differences between the time when a bug report was closed and the bug fixing commit*

Table 4.26 shows the results of comparing the distribution of time to fix a bug in the source code and in the BTS. This is specifically focused on $H_{0,0}$. As seen, the null hypothesis: "*Time to fix a bug obtained from the source code and from the bug tracking system follow the same distribution*" can be statistically rejected. The only remarkable results is the comparison between the distributions of time to fix a bug for the Xforms project. However, the coefficient for the Kolmogorov-Smirnov for this specific results was 0. This implies that this specific result should be ignored since the K-S coefficient is too

Project	K-S raw	K-S log
Actionmonkey	7.54e-231	3.94e-191
A. Tamarin	3.07e-03	1.93e-2
Camino	1.33e-164	2.31e-71
Chatzilla	0.0	4.94e-240
Comm Central	0.0	0.0
Dom Inspector	2.03e-219	1.34e-54
Graphs	1.06e-45	4.13e-14
M. Browser	0.0	0.0
M. Central	0.0	0.0
Penelope	3.52e-17	1.02e-04
T. Central	1.97e-16	2.03e-08
T. Redux	0.0	0.0
T. Tracing	8.89e-28	2.34e-13
Venkman	6.78e-206	4.39e-65
Xforms	**1.0**	1.41e-65

Table 4.26: *Study of the p-values when comparing the distributions of fixing bugs in the source code and closing bugs in the BTS using the Kolmogorov-Smirnov approach. The left column represents the results when comparing the raw values. The right column represents the values when comparing the log transformed of the raw distribution.*

low.

In addition, table 4.27 shows the result of comparing the normality or log-normality of the datasets. In this case, the hypotheses $H_{0,1}$, $H_{0,2}$, $H_{0,3}$ and $H_{0,4}$ are treated. In this case, each distribution has been tested with a Kolgomorov-Smirnov statistical analysis against a normal distribution of the data. Results show that in all of the cases the normal distribution of the datasets can be rejected. In the specific case of $H_{0,1}$: "Time to fix a bug obtained from the source code follows a normal distribution" and $H_{0,3}$: "Time to fix a bug obtained from the BTS follows a normal distribution", both can be rejected.

Regarding to the log-normal distribution of the dataset, the hypothesis $H_{0,2}$: "Time to fix a bug obtained from the source code follows a normal distribution" can also be rejected. Finally in the case of the study of the log-normality of the distribution of the time to fix a bug in the BTS, three specific cases were observed where the p-values are higher if compared to the rest of the projects. The only case where the hypothesis $H_{0,4}$ can not be rejected is in the case of the Venkman project.

4.3.6 Discussion

The first discussion point of the study is based on the history found in each repository of information. The lifespan of each project varies from 1 year (Actionmonkey) to most than 10 years presented by the Camino project. Thus, strong direct implications are found between the length of the bugs in the project and the length of history analyzed. In the

Project	N (SCM)	N (BTS)	L-N (SCM)	L-N (BTS)
Actionmonkey	0.0	0.0	0.0	6.52e-61
A. Tamarin	4.51e-27	2.49e-09	0.0	1.35e-03
Camino	1.93e-07	1.44e-49	0.0	6-13e-03
Chatzilla	9.04e-57	9.42e-274	0.0	2.04e-17
Comm Central	0.0	0.0	0.0	2.43e-18
Dom Inspector	1.77e-18	7.44e-97	0.0	1.51e-02
Graphs	2.87e-14	4.25e-23	0.0	2.30e-07
M. Browser	0.0	0.0	0.0	5.34e-57
M. Central	0.0	0.0	0.0	0.0
Penelope	2.69e-03	2.66e-08	0.0	4.36e-03
T. Central	1.34e-19	1.07e-13	0.0	**2.46e-02**
T. Redux	0.0	0.0	0.0	1.11e-243
T. Tracing	2.89e-33	1.72e-27	0.0	**5.6e-02**
Venkman	1.21e-26	3.41e-44	0.0	**0.76**
Xforms	0.0	4.56e-31	0.0	4.21e-07

Table 4.27: *Study of the p-values when comparing the distributions of fixing bugs in the source code and closing bugs in the BTS using the Kolmogorov-Smirnov approach against a normal distribution. N indicates the study of the normality of the distribution and L-N the log-normality of the distribution.*

case of Actionmonkey, it is expected to find lower times of fixing bugs if compared to the rest of the project.

Figure 4.10

In second place, it is necessary to clarify the peculiarities of the existence of several bug seeding commits per bug fixing commit. As will be seen later, in most of the cases the relationship between bug seedings and bug fixings is 1-1. However, in around a 30% of the changes, more than one bug seeding commit is linked to their bug fixing commit. This forces each study to specifically details the way the results are calculated. As seen in table 4.14 and table 4.15, depending on the method selected, the results vary.

Finally, the use of the BTS as a tracking tool has been extensively used in FLOSS projects. However, there are some limitations that have not been included in the study. In first place, although a bug owns the stage of "FIXED" or "RESOLVED" this does not directly imply that this is the original bug. In some cases this may have been a "DUPLICATED" or "WON'T FIX" bug. However, this is out of the scope of the methodology proposed.

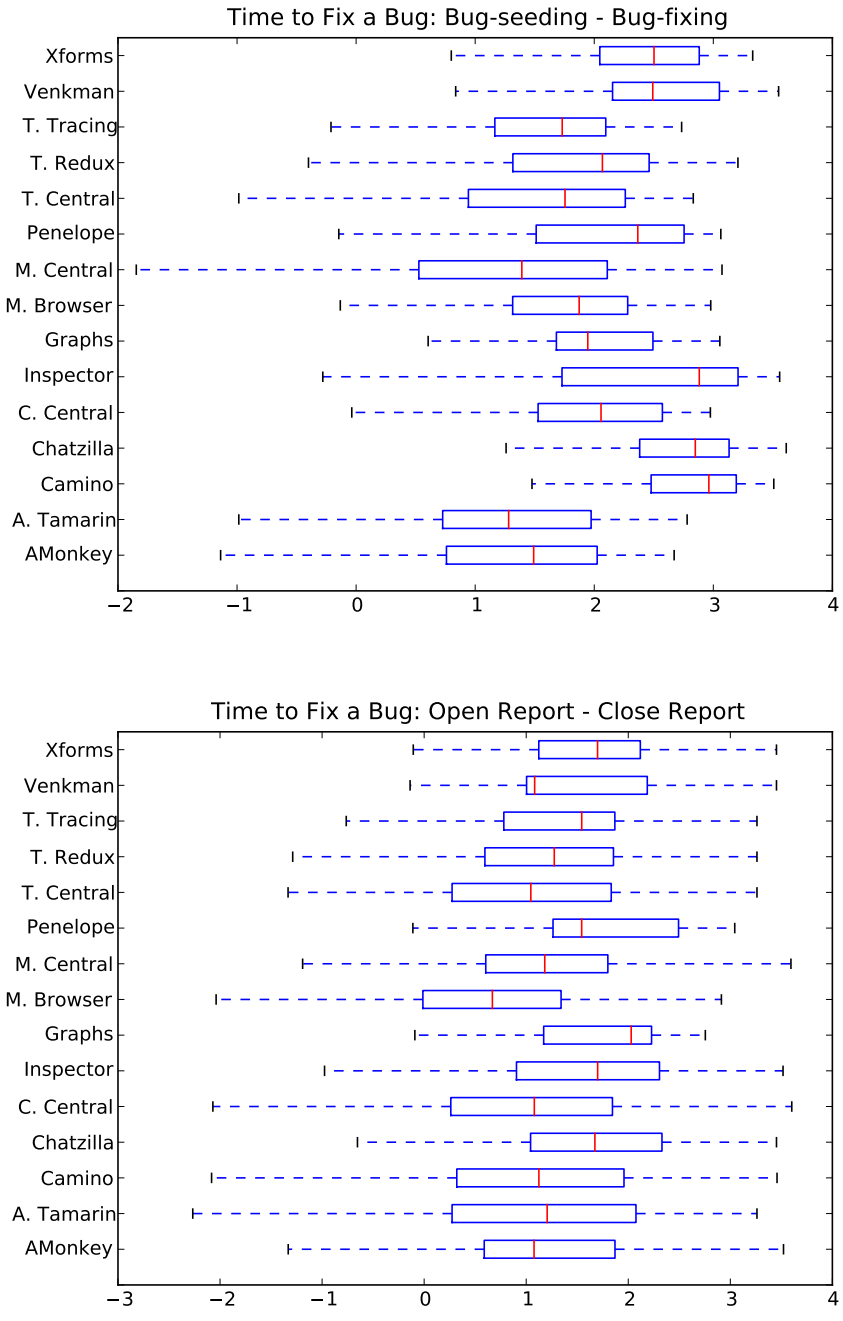

Figure 4.10: *Density charts showing the differences between the time when a bug report was closed and the bug fixing commit*

4.4 Demographical Study of the Life of the Bugs

The time to fix a bug issue has been studied from several perspectives in section 4.3: time to fix a bug in the source code or BTS, distribution of the time-to-fix-a-bug values and others. This section aims to deepen in the study of the time to fix a bug, but using another approach: *how quick are the bugs being discovered?* (or in other words: how quick are being discovered the bug seeding commits).

This study follows two main goals:

1. It has been observed how there are bugs whose time to fix takes months, even years. However, how many of them are lasting so much?.

2. This will help to answer questions to estimate when a specific percentage of the bugs (in a given period of time) have been discovered and fixed. For instance, how long does it take to fix a 50% of the bugs of the last year? The study of the distribution that is following those discoveries will answer these type of questions.

The methodology used is detailed in chapter 3, section 3.4.3. This is based on the study of the populations of remaining bugs. This approach can be seen as an analogy of the traditional demographical studies divided by gender. The calculation is made month by month (aggregating all of the bugs found in a month) and studying their evolution.

Regarding to the effect of finding several bug seeding commits per bug fixing commit, each couple of bug seeding and bug fixing commit is separately studied. The reason for this is that other types of calculation would not fit with this study. In the case of the average, as an example, it would not be realistic to show that a bug is only six months old, while the first of them is more than one year. In the rest of cases the reasons are similar, either the oldest or the youngest were selected, the rest of the bugs would not take part of the dataset.

Finally it is worth mentioning the fact that depending on the community, there is more history available to be studied. For instance, the *Camino* project contains history from 1999, but the Mozilla Central project has only got history from 2007 (when the SCM was migrated). Given that the old repository was not added to the dataset, these differences should be taken into account when discussing about the results.

4.4.1 Demographical Study: Pyramids of Population

In previous studies, it has been observed how the time to fix a bug goes from some minutes to even some years. The addition of the bug seeding commit event has added even more time if the time when that bug was *seeded* is analysed. In fact, it was measured from a 60% to a 90% of the total life of a bug the process since this *seeded* until this is *discovered*.

However, the distribution of the time to discover a bug has not been studied yet. The addition of a demographical study may help in this process where the typical pyramids of population could explain (visually speaking) if old bugs tend to remain for a long time. In any case, this will help to better understand the population of bug seeding commits and the time it takes until they are discovered.

As an example, figure 4.11 shows a demographical study of each of the projects of the case study. The selected interval is of one month (the bug seeding commits detected in one month are aggregated) and the total life represented is 30 months. Thus, the oldest bugs will be 30 months old, while the youngest will be only one month. The age is represented in the Y-axis, while the X-axis represents the number of remaining bugs for that age.

The use of 30 months has not been randomly selected for graphical reasons. If the selected age had been too young, the pyramid would have shown few cases and the evolution of the bugs would not have been appreciated. However, on the other hand, if the age of the project had too close to the final date of study, most of the bugs would have been disappeared. The explanation for the last case is based on the selection of the sample itself: along this dissertation, the bug seeding and fixing activity is studied. The existence of a bug seeding commit is always linked to a bug fixing commit. Thus, during the last month of study, most of the previous seeding commit would have disappeared since at the very end of the life of a project, all of the bug fixing commits would have taken place.

The charts show a different behaviour depending on the project. Three main sets are observed:

1. Those projects that present a traditional pyramid: this is the case of *Mobile Browser*, *Tamarin Redux* and *Mozilla Central* (the latter ignoring the 30th month). These projects present a large amount of remaining bugs in their first month of life, and during the life of these bugs, those tend to be discovered and *disappear* once they are fixed.

2. Those projects that present an inverted pyramid: in this case, the oldest bugs tend to remain for a long time. This is the case of *Camino* or *Xforms*.

3. The rest of the projects: in this group, those projects without a well defined figure, such as *Chatzilla* or *Comm Central*. In this case, there is not a defined (inverted or not) pyramid and there is some similar distribution of the bugs along the 30 years.

Checking the pyramids of population, it seems to exist a correlation between those projects named as *core* by the community, and the others. In this case, the most close to a real pyramid shape are those which take part of that *core* group (except the *Comm central* project.

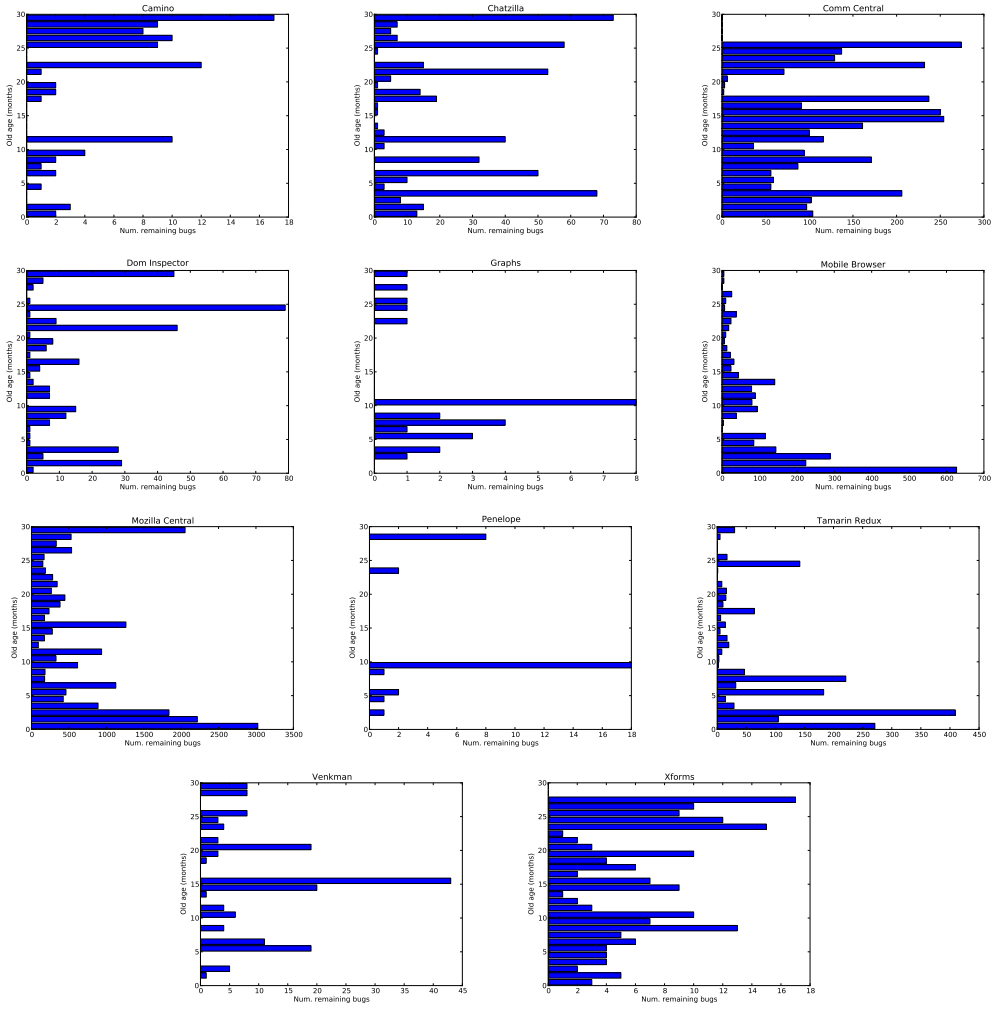

Figure 4.11: *Demographical study of the remaining bug seeding commits to be discovered in the source code. The life of each of the projects is 30 months old.*

4.4.2 Estimation of Finding Rates

As seen in figure 4.11, depending on the project, a characteristic pyramid of population is found between two extremes: typical and inverted pyramid. However, these figures are static situations in a given point in time (specifically 30 months of life) and its evolution is not taken into account.

A potential evolutionary study of the demographical study is to analyze how quick the bugs are being found. This could help in better understanding how the bugs in a specific period of time tend to decrease (or how good are being the maintenance activities).

Thus, the goal of this subsection is to calculate how quickly the remaining bugs are being found and check if they follow any type of distribution. Once it is known the distribution of those, it would be easier to predict remaining bugs based on previous data.

An initial bunch of results (508 life studies) shows that with a lineal and exponential

function it is possible to explain more than a 70% of the total finding of bugs. Around a 60% of them shows a lineal decrease, while the rest of them shows an exponential decrease.

Figure 4.12 shows an example of this. In this case the *Mobile Browser* and *Mozilla Central* project.

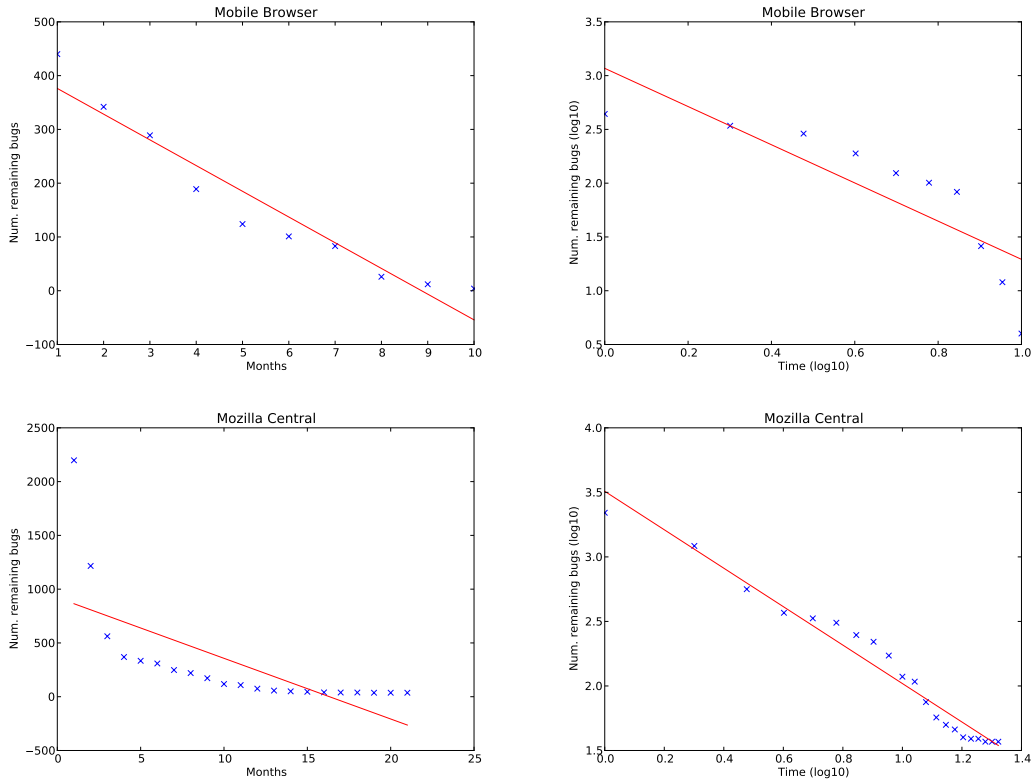

Figure 4.12: *Example of the evolution of remaining bug seeding commits. The left charts represents the net values while the right charts represent the log10 values.*

Table 4.28 shows the several results per project. Each of the numbers are the evolution for a given month of the bugs. Those charts with less than 10 points were removed due to a lack of actual results when calculating the resultant regression line. In addition to this, those projects with a bigger dataset also indicate that there is a longer history to be analyzed.

Looking at table 4.28, those projects that were defined as *core* project by the Mozilla Foundation tend to fit in the two regressions studied: lineal and exponential. On the other hand, projects such as *Camino* or *Chatzilla* tend to show a bigger number of unknown decrease distributions in general.

In order to study a bit more that 30% of the distribution which is unknown (it is worth mentioning that from those previous number, some of them were filtered and are located in the "others" column. The filter was applied to those where the value of R^2 was equal to 0, meaning that there was not actual data. In fact, it will be seen later how there are

Project	Lineal	Exponential	Others	Total
ActionMonkey	5	0	2	7
ActionMonkey Tamarin	3	2	1	6
Camino	26	0	38	64
Chatzilla	64	11	19	94
Comm central	15	2	5	22
Dom Inspector	21	5	47	73
Graphs	5	5	5	15
Mobile Browser	20	5	2	27
Mozilla Central	6	25	12	43
penelope	2	1	4	7
Tamarin Central	5	4	1	10
Tamarin Redux	33	8	5	46
Tamarin Tracing	4	3	1	8
Venkman	9	9	19	37
Xforms	13	12	17	42

Table 4.28: *Distribution followed by the set of one-month-bug-seeding-commits through the time.*

bugs that remain for a long time)).

In all of the cases of the unknown values, it has been detected that old bugs simply remain in the source code for a long time and they do not even decrease. Thus, the demographical study would show the same numbers month by month.

An example of this can be seen in figure 4.13 where a chart has been added as a representation of this issue. The left chart shows the aforementioned behaviour where there is not a real decreasing function. The right chart represents a typical decrease in the evolution of the selected values. In the case of the left chart, that set of bugs is included in the column of *Others* distributions.

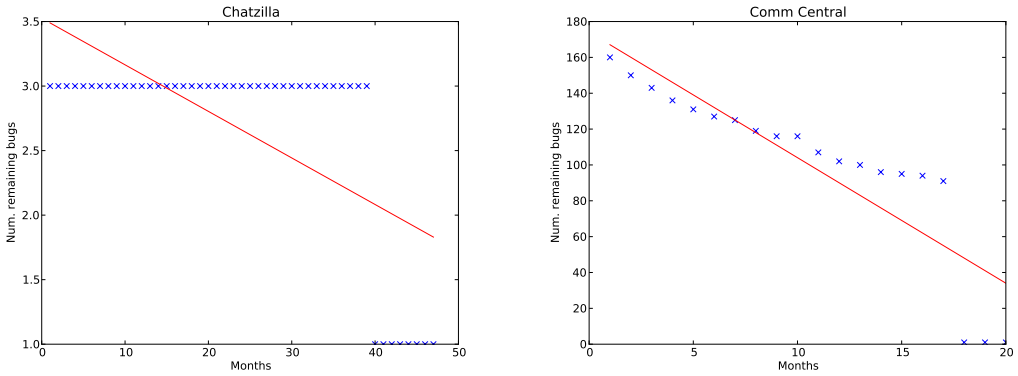

Figure 4.13: *Evolution of bug remaining bugs. The left chart represents a set of bugs that does not evolve. On the contrary, the right chart represents a set of bug seeding commits that are found and decrease month by month.*

Thus, for specific projects, it is possible to ask ourselves specific questions that may be important from a cost or effort estimation point of view: when is at least a 75% of the bugs for a given month/year (or any other period we would like to define) going to be found? How many months do we need to wait to find a 50% of the bugs that were introduced at the beginning of the year? The answer to this question could be easily answered once the distribution of the evolution of bugs is known. Although not specific studies have been carried out in this dissertation, those could be easily obtained.

Goodness of Fit.

Finally, in the following figures, there is a representation of the goodness of fitness of this approach, where the R^2 of the several datasets is studied. In first place, figure 4.14 depicts the goodness of fitness of the whole dataset if this is adjust to a lineal model (left) or to an exponential model (right).

The lineal model seems to represent better the general fitness of the population, since around a 50% of the total values are over a R^2 of 0.8. In fact the median value is 0.82. On the other hand, in the exponential model, the values over 0.8 represents less than a 25% of the whole dataset, being the median 0.66. In any case, both representations show that there is a higher density of values at the top side of the charts.

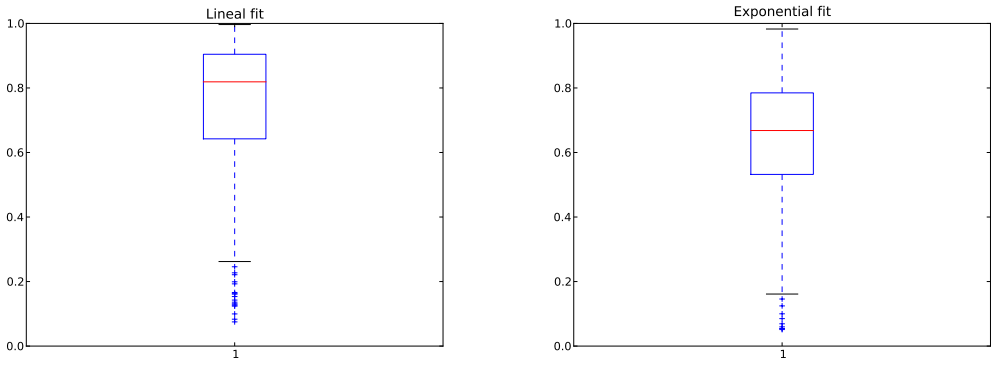

Figure 4.14: *Goodness of the fitness for a lineal and exponential models*

In second place, figure 4.15 depicts the goodness of fitness of the whole dataset if this is adjust to a lineal (left) or exponential (right) model. Depending on the better value of the R^2, this is decided to be included in a lineal or exponential model. For example, if there is a R^2 of 0.99 for a lineal model and a 0.85 R^2 for an exponential model, the lineal model is taken as better fitness than the exponential model. In addition, the results show how the median of the lineal model is around 0.9 and in the exponential model, the median is around 0.86. This, again, show how, in general, the exponential model (even for those values that are better in the exponential model) that there is a lower rate of goodness when fitting those values.

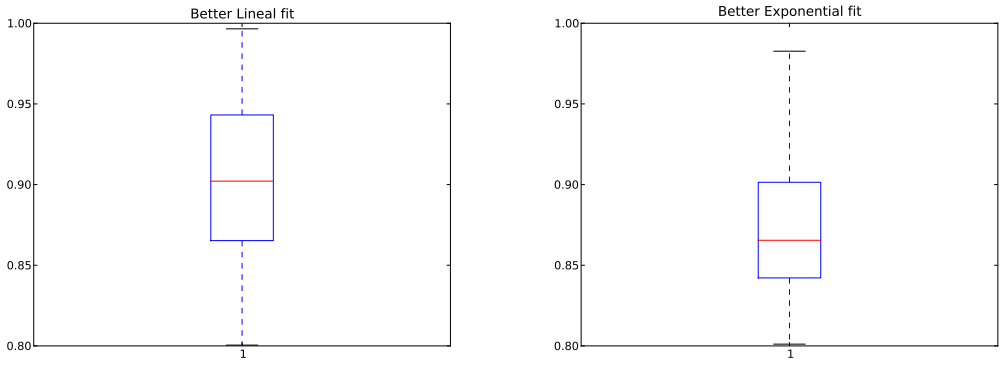

Figure 4.15: *Goodness of the fit for a lineal and exponential models (over 0.8 for R^2)*

Regarding to the limitations of this study: first of all, the statistical approach is far from being perfect. Although 30% of the total amount of data has not been studied, and not included when estimating number of remaining bugs, it is possible to make an in depth approach using a survival analysis technique. In fact, the use of the regression line is supposed to be used when two variables are independent. However, in this case, one variable is totally dependent from the one behind it. Or in other words: when the number of bugs is decreasing, the next month depends from the current one. Another limitation of the study is the use of regression line when one of the variables is the time.

However, since we are only interested in studying the distribution of the decreasing function, it is more than enough to study the resultant regression line, since in this way, if month by month the number of bugs remaining decrease in a lineal way, this is seen in the regression line by means of the R^2 value and how they well fit to the general approach used.

4.5 Relationships between Bug Seeding and Bug Fixing Commits

The goal of this section is to study the relationship between bug seeding and bug fixing commits. The literature has raised interesting questions related to the software maintenance field, such as: the *defective fixing changes* [Kan, 2002] or number of different people involved in a specific section of the source code to predict bugs [Nagappan *et al.*, 2008].

Thus, this section aims to study the relationship between bug seeding and fixing commits from the two aforementioned perspectives: how many bug seeding commits are related to a bug fixing commit and how many bug fixing commits are introducing new errors in the source code.

4.5.1 Bug-fixing Commits which are a Potential Bug Seeding Commit

According to [Kan, 2002], there are specific metrics that haven been listed to track how well the maintenance process is being carried out. During the maintenance phase several defects may be detected and those will be solved by the community. However, the authors claim that: "... *the number of defect or problem arrivals is largely determined by the development process before the maintenance phase* ...".

However, it can be claimed that in FLOSS projects the maintenance and development phases are a bit fuzzy sometimes [Capiluppi *et al.*, 2007], even more when a project may have been abandoned and later recovered from another set of developers. In addition, the *"release early, release often"* assumption that usually is taken in FLOSS communities may distort this approach.

In any case, bringing in context a client's perspective, the finding of errors in the execution of the program is something unexpected that could directly influence in the business. In the specific case of FLOSS, instead of affecting the final perception of a product and the reputation of the development company, this would affect the reputation of the community and the selection of another product similar in functionality. Thus, the process of finding bugs or at least knowing in advance about their existence is a key factor to be taken into account.

Thus, the introduction of errors in the source code is something that commonly happens and it is almost impossible not to introduce them when a software is evolving. In fact, although the goal of the maintenance phase is that: *"... the quality goal for the maintenance process, of course, is zero defective fixes without delinquency ..."* [Kan, 2002][10]. It is hardly impossible to find a maintenance process with those characteristics.

[10]Delinquency is defined as those issues detected by the costumer that has exceeded the pre-defined time to be fixed depending on the severity.

Specific comments on the contrary can be found at [Lehman, 1979 1980] based on Lehman's laws and related literature [Endres & Rombach, 2003].

One of the metrics that are used to estimate how well a maintenance process is being undertaken is the already introduced *defective fixes* metric. This metric is calculated as the percentage of the fixing commits that in the future are detected to introduce a new error in the source code. Table 4.29 shows the results per project for this metric. It is detected main differences in terms of defectiveness of the commits fixing a previous error. Only taking those projects with a number of analyzed commits higher than 1,000, those community-driven (Mozilla Central and Comm Central) show the highest number of fixing commits that are later defective. On the other hand, those company-driven communities, such as ActionMonkey or Tamarin Redux the ratio is lower.

However, it is worth to remember that the ActionMonkey project was abandoned after a year which may distort the results. All in all, the project Tamarin Redux, where people from Adobe are the core developers (as will be seen in later experiments) seem to reach only a 54% of defective fixes.

This may open a new set of questions related to the quality of the changes depending on the leaders. However, another point of view could be used and this is that the FLOSS communities commanded by volunteers are more dynamic, and this provokes these kind of results.

Finally, also the stage of the project may vary these results. For instance, the stage of development for the Comm Central community is not the same as for the Mobile Browser community where the latter is much younger and probably it is closer to a development phase than to a maintenance phase.

As a summary, all of the projects show a ratio higher than a 25% of defective fixing commits (for each four fixing commits, one of them is introducing new errors in the source code). And probably, the most significant of them, with a ratio higher than a 50% are introducing for each couple of bug fixing commits a new error in the source code.

4.5.2 Number of Bug Seeding Commits per Bug Fixing Commit

As specified in the introduction of this section, this is studying the relationship between bug seeding and bug fixing commits. In the previous subsection, fixing commits were studied and checked how many of them were detected as causing fixing actions.

This subsection aims to study the other way around: how many bug seeding commits are related to a bug fixing commit. By definition of bug-fixing commit, there must be at least one associated bug seeding commit. Besides, by definition, several bug seeding commits may appear as being part of a later bug fixing commit. Thus, a first study will help us to answer question related to the number of previous changes involved in a new error.

Projects	Defective fixing commits	Total fixing commits
Actionmonkey	2,160 (58.00%)	3,724
Actionmonkey Tamarin	56 (62.92%)	89
Camino	34 (**27.64%**)	123
Chatzilla	315 (58.65%)	537
Comm Central	2,446 (**93.21%**)	2,624
Dom Inspector	146 (73.73%)	198
Graphs	27 (55.10%)	49
Mobile Browser	1,163 (78.58%)	1,480
Mozilla Central	23,216 (88.25%)	26,305
Penelope	12 (44.44%)	27
Tamarin Central	41 (51.25%)	80
Tamarin Redux	826 (54.34%)	1,520
Tamarin Tracing	50 (71.42%)	70
Venkman	57 (52.77%)	108
Xforms	31 (32.97%)	94

Table 4.29: *Bug fixing commits detected as being defective (that later were detected of introducing a new error in the source code).*

In order to visually observe this phenomena, the density charts of the relations between bug seeding and fixing commits is shown in figure 4.16. As aforementioned, there could appear some fixing commits whose seeded process was done by several commits. Generally speaking, for all of the studied projects, more than a 50% of the commits were previously handled in just one bug-seeding commit

Indeed, these results are more extreme in projects such as *Mozilla Central* where a huge quantity of the errors were introduced in only one previous commit. This also means that for a given set of lines, in most of the cases (more than a 50% of them) those lines that have been removed or modified, were previously added or modified as they are.

For instance, this result may be useful from the program slicing point of view: understanding how the process happens when a bug is introduced in the source code, we may learn how this bug evolves over time and remain in the source code. Thus, if this set of lines is studied as they are, it may help when a bug is found in the source code.

This opens another research question related to how many people tend to introduce issues in the source code. And how many of them realized of their own errors and fix them. A paper with this respect has shown that only a 5% of the total *corrective* changes in the source code were made by the same developers than introduced the issue [Izquierdo-Cortazar *et al.*, 2012].

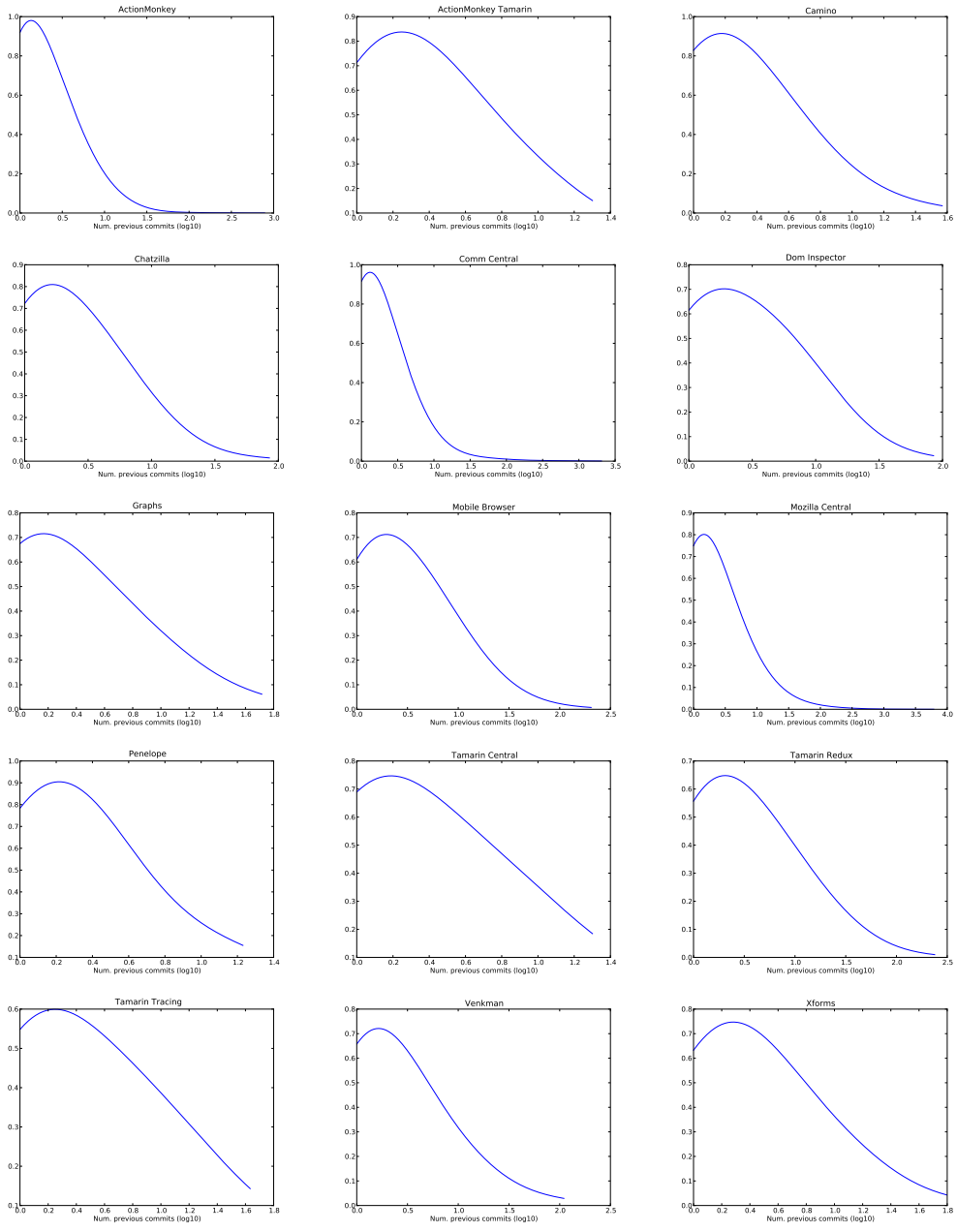

Figure 4.16: *Number of previous commits involved in the seeding process of a bug*

4.6 Bug Seeding Activity and Experience

Up to here the bug life cycle has been studied from several perspectives. The following sections will analyze the potential impact of human factors that may alter the *quality* of the changes.

Specifically, this section will study the relationship between *experience* and the *quality of the changes* measured by means of the number of *buggy* changes (bug seeding commits).

As seen in chapter 2, experience is measured in the literature in different ways. However, from a quantitative point of view, three main paths have been detected: *number of commits*, *number of fixing commits* and *ownership* of the source code.

Regarding to the first of them, number of commits, this is the usual way of couting commits. With respect to the second one, number of fixing commits, those are detected, as seen in chapter 3, section 3.3.1 by means of the messages left by developers in the SCM. Finally, the third one is obtained through the study of the *territoriality* metric that is calculated as the number of files touched by just one developer (as seen in chapter 3, section 3.4.4).

Regarding to the way *quality* of the changes is measured, those will be divided between the ones that did not introduce any error in the source code and those that did introduce an error. Mixing both variables, the concept of *bug seeding ratio* is obtained. This outputs a percentage and it is calculated as the number of bug seeding commits out of the total activity. A similar concept has been previously used measuring activity carried out by developers [Eyolfson *et al.*, 2011] and dividing the commits between *cleany* and *buggy* [Kim *et al.*, 2008c]. This concept can also be seen from an evolution point of view, where the bug seeding ratio evolves thanks to the types of commit (clean or buggy) done by a developer.

In general, based on the previous concepts, the following research questions will be addressed:

1. **Analysis of the bug seeding ratio**: the goal of this research question if focused on the study of the bug seeding ratio as a phenomena and its potential evolution through the time for a given project or developer. Using as an approach the fourth and fifth Lehman's Laws, the bug seeding ratio should increase since the complexity of the source code increases. This will be studied in section 4.6.1.

2. Once this is understood, the **relationships between experience and the bug seeding ratio** is studied. As detailed, the experience can be measured in different ways and all of the quantitative related ways will be correlated against the bug seeding ratio. It is not well understood if the expertise in a project is directly related to the less introduction of buggy changes. From an intuitive point of view, it is expected to find that the less experienced developers will introduce more buggy

ID	Null hypothesis	Test applied	Attribute
$H_{0,0}$	The bug seeding ratio of the developers follows a normal distribution	Anderson and D'Agostino	bug seeding ratio
$H_{0,1}$	The older the project, the higher the bug seeding ratio	Correlation	age and bug seeding ratio

Table 4.30: *Null Hypotheses*

changes. This will be studied in section 4.6.2.

3. **Working at the granularity of developers**: as detailed in the methodology section, it is possible to calculate the several types of commits that a developer usually undertakes. Thus, based on the differentation of bug fixing and bug seeding activities, it is possible to make a characterization of developers. It is expected to find set of developers mostly focused on maintenance activities. This will be studied in section 4.6.3

4.6.1 Introduction to the Bug Seeding Ratio

The bug seeding ratio is calculated as the number of seeding commits out of the total activity carried out. The outcome is a percentage that determines a relative number of "buggy" changes submitted to the source code management system.

This relative number will help to better understand how the bug introduction process is spread along a project. Some people will be found to be introducing more or less buggy changes. And all of this data, if aggregated, will provide a first glimpse about the general distribution of bug seeding ratio among developers and the typical bug seeding ratio in a project.

In addition, the evolution of the bug seeding ratio can be studied in order to analyze the evolution of buggy changes in the source code if compared to the total activity. This will help to understand if a project is increasing its buggy changes. This is also applicable at the granularity of developers.

Table 4.30 shows the null hypotheses to be tested.

Hypotheses testing $H_{0,0}$: The bug seeding ratio of the developers follow a normal distribution.

Figure 4.17 shows the bug seeding ratio divided into several discrete values. Besides a filter of at least 10 commits has been added in order to avoid including too much noise to the results. Without that filter, most of the charts would have shown two big bars close to the 0% and 100% of bug seeding ratio. This behaviour is close related to the fact that loads of developers only submit one or two commits in their whole life (for instance, fixing an issue and uploading a patch to the bug tracking system). Thus, with only a couple of

commits, there are a lot of possibilities that all of the commits are either buggy or clean.

In addition, the biggest projects in terms of number of developers, such as ActionMonkey, Comm Central or Mozilla Central tend to show a normal distribution (this hypothesis will be later analyzed). On the other hand, some other projects tend to show a more uniform distribution of bug seeding ratio (in most of the cases due to the low number of committers). Finally, those projects that do not appear in the charts were removed due to their small size.

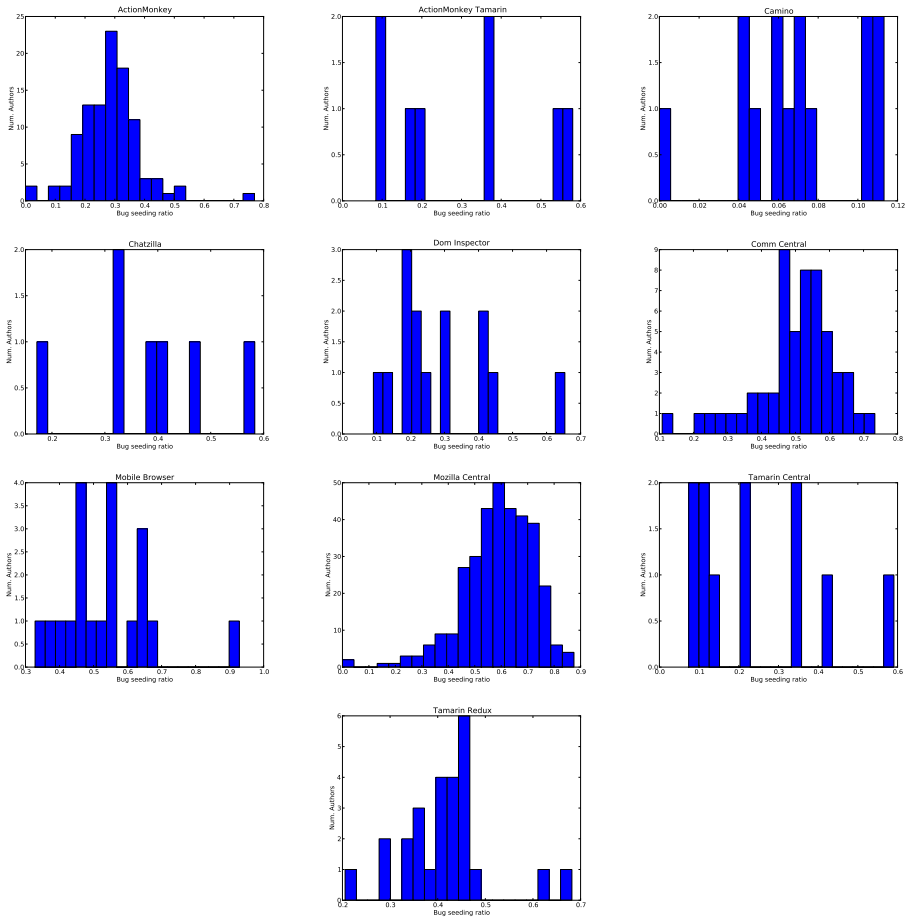

Figure 4.17: *Bug seeding ratio histograms: percentage of bug-seeding commits per developer and aggregated by project*

The hypothesis $H_{0,0}$ aims to test if the bug seeding ratio values, visualized at figure 4.17 follow a normal distribution. For this purpose, three statistical approaches have been use to estimate the normality of the distributions: Anderson, D'Agostino and Kolmogorov-Smirnov tests.

Values over 0.972 in the Anderson coefficient test implies the rejection of the null hypothesis. In all of the cases, the critical values are limited by the array of values: $array[0.513, 0.584, 0.701, 0.817, 0.972]$ for the following significant levels specified for a

Project	Anderson test	D'Agostino test	Kolmogorov-Smirnov test
ActionMonkey	1.28	1.51-e06	0.0
ActionMonkey Tamarin	**0.37**	**0.43**	**1.18-e02**
Camino	**0.32**	**0.64**	9.24-e04
Comm Central	1.02	8.11-e04	0.0
Dom Inspector	**0.55**	**7.77-e02**	2.72-e04
Mobile Browser	**0.44**	6.26-e03	3.54-e08
Mozilla Central	2.066	3.99-e13	0.0
Tamarin Central	**0.51**	**0.23**	2.11-e03
Tamarin Redux	**0.73**	**8.65-e02**	7-11-e09
Tamarin Tracing	**0.25**	**0.97**	1.25-e03

Table 4.31: *Study of the normality of the bug seeding ratio distributions*

normal or exponential distribution: $array[15\%, 10\%, 5\%, 2.5\%, 1\%]$. For the rest of the test, the *p-value* is provided. Table 4.31 shows the results. The values whose null hypothesis is not possible to reject are in bold.

The results show that the values are not normaly distributed in most of the cases. While the Anderson tests indicates that in most of the cases the null hypothesis can not be rejected, the Kolmogorov-Smirnov test indicates the opposite: the null hypothesis can be rejected with a confidence of a 98%. These results could be bias by the limited number of values found in each of the projects. As seen in figure 4.17 most of the projects show a pretty low number of values to be analyzed what may imply that distortion of the results. Regarding to the biggest projects: ActionMonkey, Comm Central, Mobile Browser, Mozilla Central and Tamarin Redux, the results show that the null hypothesis can be rejected. Thus the bug seeding ratio does not follow a normal distribution.

In addition, another discussion could be opened related to the uniformity of the results. Visually speaking, it has been observed how the distribution of the bug seeding ratio is not uniformly distributed. Although this results was expected, the observation of the charts show how there are specific ranges of bug seeding ratio where developers tend to concentrate. Besides, those ranges are shifted to the right or left depending on the project.

This concentration of the bug seeding ratio along a specific range and also the potential normal distribution of the dataset in some others, may be explained by two potential causes: in first place the accuracy of the results when using the log message left by developers. There may be communities where this policy is not so extended as in others. This may provoke an increase of the bug seeding ratio in those communities where developers usually tend to specify if a commit is fixing an issue. And, in second place, the results may be explained as a difference between young and more mature projects. It could be argued that more mature projects usually follow the general policies more strictly than others.

In any case the results show that after some time of activity (in this case around four years) there is a concentration of bug seeding ratio among a given range. Once this range is known, it is easily expected to reach certain level of buggy commits. Together with a more strict policy when indicating fixing bugs in the source code, may help to better understand the peculiarities of each community inside the Mozilla Foundation.

Analysis of the Evolution of bug-seeding ratio

As said at the beginning of this subsection. Developers may be affected by external factors forcing them to behave in another way as usual. For instance, communities could agree with using a new policy regarding to use a standard when writing down the message log. Thus, this analysis aims to check if there is a stability in this ratio and checking if there may appear external factors that may alter the stability of the bug seeding ratio. This would improve the general software development process of the community from two perspectives: fixing actions are clearly identified and this would help when detecting the origins of those fixing commits. With this respect, figure 4.18 shows this evolution for each of the studied projects.

In most of the cases, at the beginning of the chart there is an initial range of noise provoked by the way these charts are calculated. For a given project, there are two sets of commits: those that have been detected as introducing an error and the rest of them. Since the bug seeding ratio is calculated by means of the number of erroneous commits out of the total activity, once there are enough values, the chart tend to stabilize. In addition, the current value is totally dependent from the previous one, since the data is being aggregated. As an example, if there is a bunch of commits that do not introduce errors, the bug seeding ratio will tend to decrease.

The results of these figures could help us to argue that in most of the projects, there is a clear relation between time and a decrease or stability in the bug-seeding ratio if compared to the typical activity of the community. However, there are others such as the Mobile Browser or Mozilla Central where this ratio tends to increase month by month.

Hypotheses testing $H_{0,1}$: The older the project, the higher the bug seeding ratio.

This hypothesis aims to test the correlation between the age of a project and the number of issues that are raised. According to the seventh law of software evolution by Lehman [Lehman & Ramil, 2006], there is a declining quality in the source code. This could be avoidable if effort in maintenance and rengineering activities are undertaken. As a definition of quality, this dissertation has used the metric of counting *buggy* commits or the relative number of the bug seeding ratio. Thus, it is expected to find a higher bug seeding ratio while the project is evolving unless any refactoring or preventive actions are undertaken. This can be partially visualized in figure 4.18 where projects such as Dom

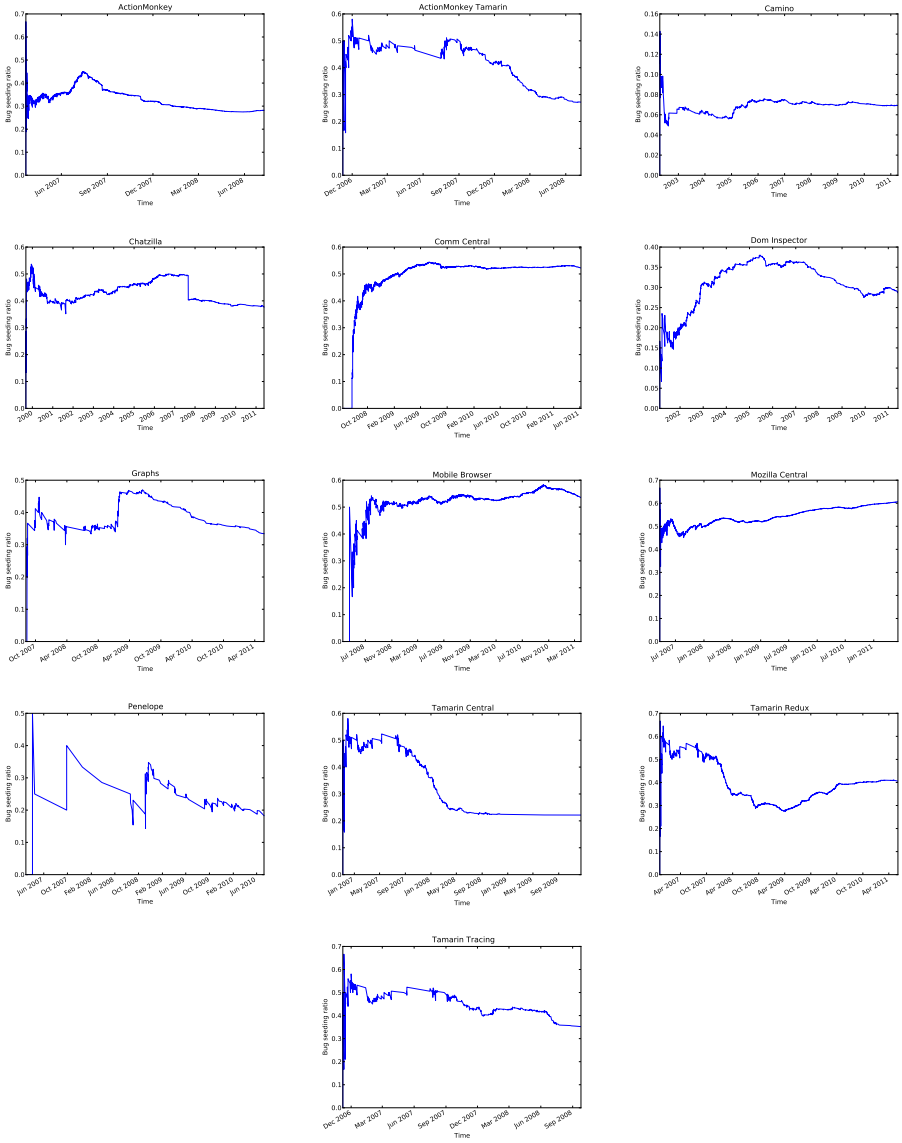

Figure 4.18: *Evolutionary study of the percentage of bug seeding ratio per project*

Inspector presents an initial increase while after 2007 this ratio starts to decrease year by year, what may indicate that the maintenance activities are working well.

Bug seeding ratio and software maintenance

Thus, the bug seeding ratio is used as a variable for this hypotheses testing. So far there are visual indications that the age of a project may indicate a better maintenance process. This is seen by the decrease of the bug seeding ratio in figure 4.18. Thus, the hypothesis $H_{0,1}$ would be rejected in some cases if correlating the trend of the bug seeding ratio and the age of the projects.

In addition to the hypothesis testing, this part of the experiment aims to use extra maintenance variables that may help to confirm the hypothesis $H_{0,1}$. A better maintenance

effort may indicate a lower introduction of new errors and a quick fix of the remaining ones. According to [Kan, 2002], there is another variable not studied so far in the dissertation that is focused on the software maintenance field. And this is used when estimating the quality of the maintenance activities. The backlog management index (BMI) is defined in the following way:

$$BMI = \frac{Numberofproblemsclosedduringthemonth}{Numberofproblemarrivalsduringthemonth} * 100\%$$

However, most of the studies focused on net number of issues usually takes the data from the bug tracking system. Since this dissertation is studying the bug life cycle, the bug seeding and bug fixing commits will be used. Thus, the definition of the BMI coefficient would be translated to the following one:

$$BMI^1 = \frac{Numberoffixingcommitsduringthemonth}{Numberofseedingcommitsduringthemonth} * 100\%$$

Besides, although the bug seeding ratio has been used as in a given point or in an evolutionary study, in this metric, to be comparable to the BMI coefficient, the bug seeding ratio will be calculated month by month. However, in this case, it is not used when a commit is detected as seeding a bug, but the number of future bugs that are seeding (a bug seeding commit can seed from 1 to dozens of bugs in the future). Thus, the definition of the BSR (Bug Seeding Ratio) coefficient is the following one:

$$BSR = \frac{Numberofcommitsseededinthefuturepermonth}{Numberofcommitspermonth} * 100\%$$

The results of the evolution of the BMI coefficient are depicted in figure 4.19 where the index is calculated month by month. Those projects with no data have been removed. A coefficient of BMI under 100 shows that there are being opened more issues than closed ones. Or in other words: the smaller the BMI index, the worse the corrective maintenance activity. In addition, the evolution of the charts allow to study the trend of the BMI coefficient. An increase in the BMI coefficient would indicate an improve in the maintenance tasks carried out by the community.

However, some limitations may be found due to the definitions of bug seeding and bug fixing commits: by definition of bug seeding and bug fixing activity along this dissertation, it is normal to find a big peak of the coefficient at the very end of the studied history. In this case, all of the remaining bugs in the source code would have been found and later fixed.

Focusing on the charts, there are several analysis that should be made:

1. In first place, **the trend of the chart**. A visually increasing BMI coefficient would mean that the maintenance activity is working according to the expected results of a maintenance activity: fixing errors and not causing new ones. At the level

of bug seeding and fixing activity, this would mean that the number of problems closed during the month is increasing and increasing if compared to the number of new issues being seeded. An increase in the BMI coefficient is seen (ignoring the very last peak) in ActionMonkey, ActionMonkey Tamarin, Comm Central, Mobile Browser, Mozilla Central and Tamarin Central. In the rest of the projects it is not clear (at least visually speaking) if the BMI coefficient is increasing.

2. In second place, **the value of the BMI coefficient**. In most of the studied cases, the BMI coefficient is found under 100%. This indicates that in each period studied (month by month), the number of new seeded issues are more than the number of fixed issues. The only projects found that usually show BMI coefficients over 100% are the Chatzilla and Comm Central.

5best maintained project from the cases of study is the Comm Central project. In first place,

In addition figure 4.20 shows the evolution of the bug seeding ratio in a discrete way if compared to the aggregation of data done in figure 4.18. As detailed, the bug seeding ratio measures the number of commits detected as introducing an error against the total number of commits. In this case, the number of buggy commits have not been used, but the number of future bugs that appear because of a buggy change. Thus, for a given change, there could appear from zero to dozens of future bugs. Although this would depend in some cases in the size of the changes, this is not studied here and it is left as further work.

Thus, the charts seen in figure 4.20 shows the total number of future bugs that will be found because of the buggy commits. A higher value represents a worse indicator if compared to a lower value. Values over 100% means that the changes done in the source code are introducing much more bugs than the typical activity per month. As an example, in the case of the Chatzilla project, there are peaks of bug seeding ratio where for each change in the source code the developers are introducing up to three future bugs. This is even more serious in the case of the Mobile Browser or Mozilla Central projects where the bug seeding ratio is reaching a 600%.

On the other hand, there are projects such as Camino where the bug seeding ratio by month only reaches a 40% of the changes what denotes a better improvement in maintenance activities.

Finally, it is worth mentioning the increase or decrease of the charts. A decrease in the charts, as seen in ActionMonkey or Comm Central, indicates a better development activity since the number of buggy changes are decreasing if compared to the total activity in the project.

Summarizing: in order to check the relationship between age and bug seeding ratio,

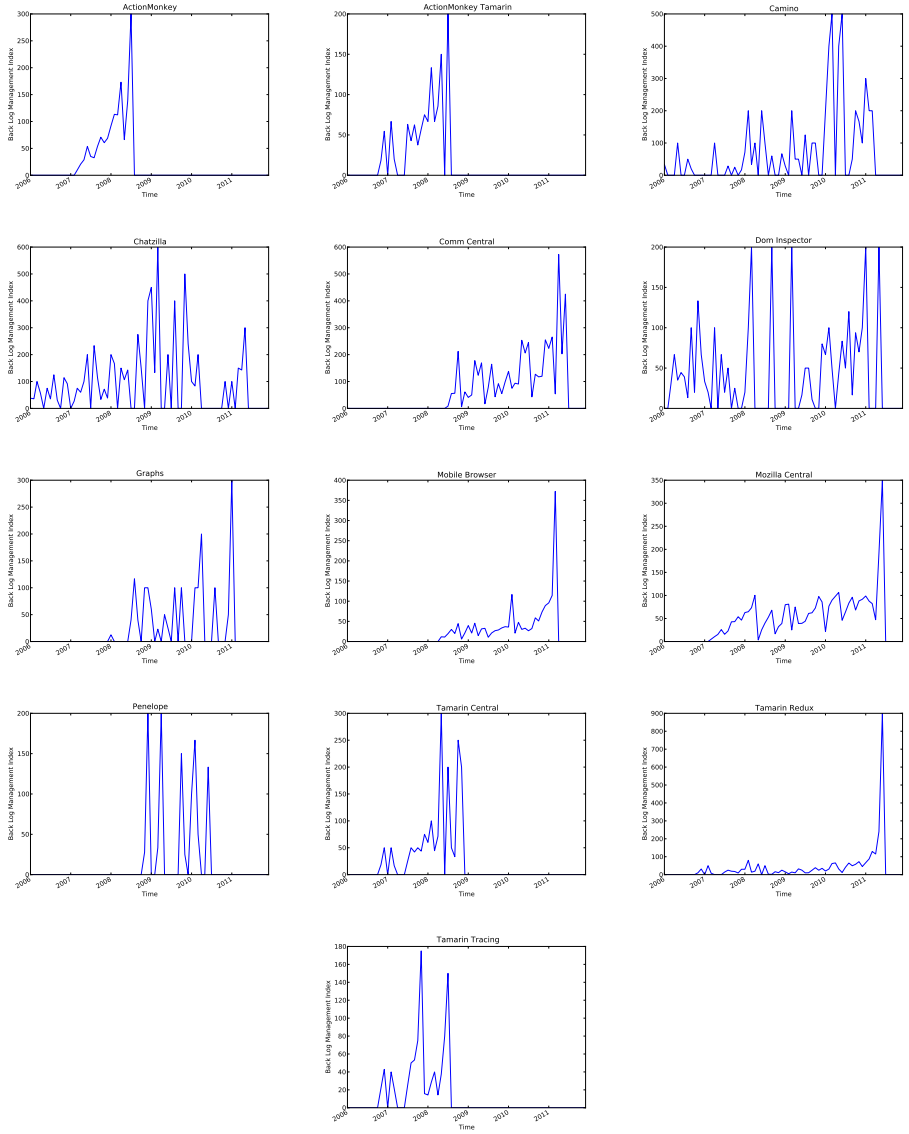

Figure 4.19: *Backlog management index coefficient: evolution month by month.*

three different experiments were done: observational study of the evolution of the bug seeding ratio, observational study of the evolution of the BMI coefficient index and observational study of the evolution of the BSR coefficient index. The results have been aggregated in table 4.32 where per each project, the tendency of the charts is shown.

The first column: *BMI coeff.* indicates the tendency of the backlog management index. As seen in figure 4.19 and detailed in the definition of this index, an increase of this represents a good effort in maintenance tasks. The second column: *BSR coeff.* indicates the tendency of the bug seeding ratio measured month by month. As seen in figure 4.20 and detailed in the definition of this index, an increase of this represents a bad evolution. This would indicate an increase in the number of future bugs that are being seeding in

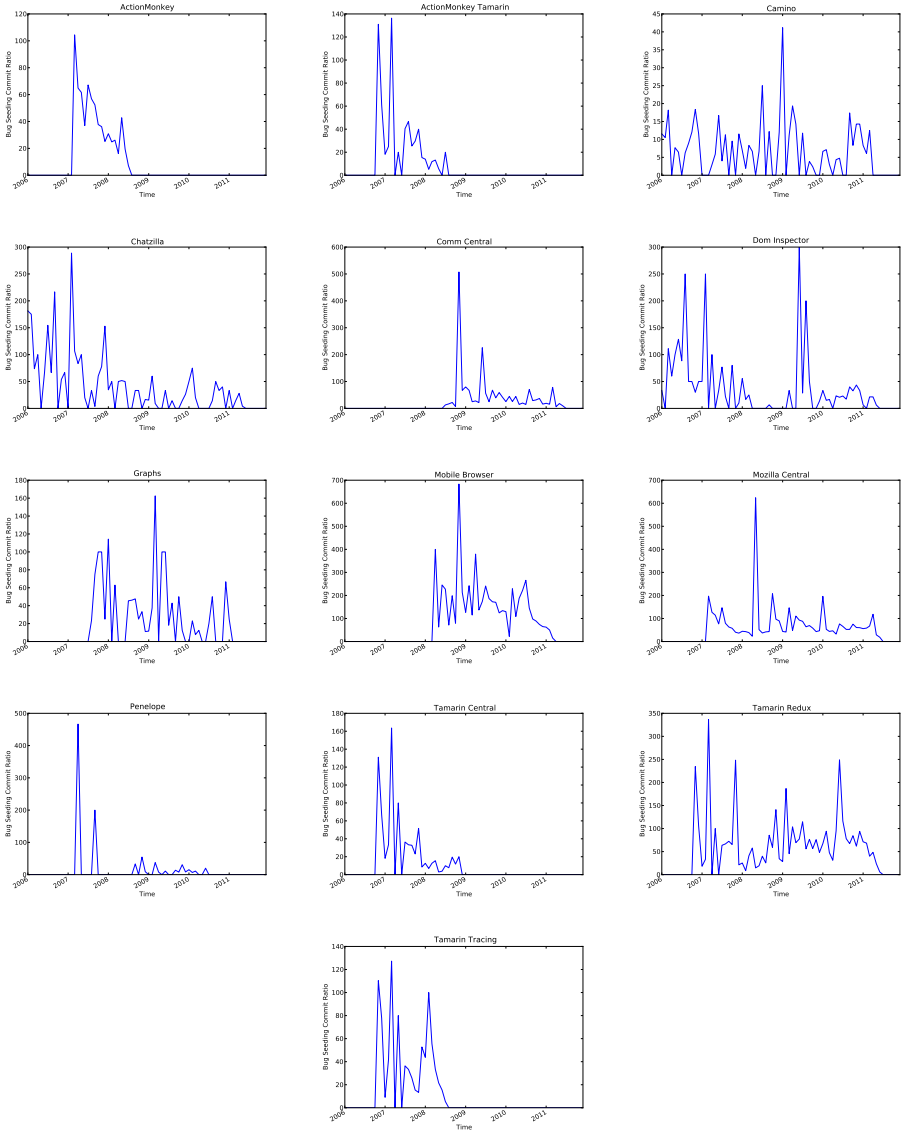

Figure 4.20: *Bug seeding ratio: evolution month by month*

each month if compared to the activity carried out in that moment. The third column: *BSR evolution* indicates the tendency of the bug seeding ratio along the history of the projects. This is represented in figure 4.18. A decrease of the chart would indicate a better behaviour since the percentage of buggy commits is decreasing out of the total activity. Finally, the fourth column represents a first glimpse about the maintenance of each of the projects based on the three metrics. This should be taken carefully since loads of other metrics have not been part of this analysis.

Discussion

As a discussion for each of the projects, those where the BMI coefficient is increasing would mean that the community is fixing more bugs than the number of bugs that are

Project	BMI coeff.	BSR coeff.	BSR evolution	Result
ActionMonkey	↗	↘	↘	
ActionMonkey Tamarin	↗	↘	↘	
Camino	↗	→	→	
Chatzilla	→	↘	↗ ↘	
Comm Central	↗	→	→	
Dom Inspector	→	↘	↗ ↘	
Graphs	→	→	↘	
Mobile Browser	↗	↗	↗	
Mozilla Central	↗	→	↗	
Penelope	→	→	↘	
Tamarin Central	↗	↘	↘ →	
Tamarin Redux	→	↗	↘ ↗	
Tamarin Tracing	→	→	↘	

Table 4.32: *Summary of the several factors used when measuring maintenance and evolution of the bug seeding ratio.*

being introduced (at least for that month). This is the case of most of the projects where this coefficient tend to increase or to remain stable.

In addition, those projects where the BSR coefficient is increasing would mean that although for each of the commits, they are introducing much more future bugs. Thus, it is compatible to find an increasing BMI coefficient and an increasing BSR coefficient. As an example, projects where both coefficients are increasing is the Mobile Browser project.

Finally, the evolution of the bug seeding ratio indicates the evolution of the percentage of the bug seeding ratio in an aggregated form. Thus, it does not care if a month is adding too many errors in the source code, but the aggregated history.

Thus, those projects that present a better evolution in time regarding to the three factors used are the ActionMonkey and ActionMonkey Tamarin. Then the projects Chatzilla and Dom Inspector. On the other hand, projects that are showing a worse maintenance and bug seeding ratio evolution are Tamarin Redux and Mobile Browser and Mozilla Central, where the bug seeding ratio is increasing. These results are specified with a color (green, yellow or red) at the end of the table.

4.6.2 Relationship between Experience and Bug Seeding Ratio

As detailed in chapter 2, section 2.3.2, the expertise of a developer can be measured taking into account specific variables. Among them, the number of commits done [Mockus & Herbsleb, 2002], [Minto & Murphy, 2007] [McDonald & Ackerman, 2000], [Girba *et al.*, 2005], ownership [Fritz *et al.*, 2007a], [German, 2004a] or fixing activity [Ahsan *et al.*, 2010].

This subsection aims to study the relationship between experience and the bug seeding

ID	Null hypothesis	Test applied	Attribute
$H_{0,0}$	There is not correlation between bug seeding ratio and number of commits	Pearson two-tailed	bug seeding ratio and overall commits
$H_{0,1}$	There is not correlation between bug seeding ratio and territoriality	(same)	bug seeding ratio and territoriality
$H_{0,2}$	There is not correlation between bug seeding ratio and fixing activity	(same)	bug seeding ratio and fixing commits
$H_{0,3}$	There is not correlation between number of commits and territoriality	(same)	overall commits and territoriality
$H_{0,4}$	There is not correlation between number of commits and fixing activity	(same)	overall commits and fixing commits
$H_{0,5}$	There is not correlation between number of territoriality and fixing activity	(same)	territoriality and fixing activity

Table 4.33: *Null Hypotheses*

ratio. As explained, the experience can be measured in different ways and they will be taken into account in the different hypothesis testing: number of commits, ownership (using the metric of territoriality) and fixing activity. In addition, it will be tested if there exist any kind of relationship between the several definitions of expertise.

Table 4.33 summarizes the hypotheses to be tested.

Hypotheses testing $H_{0,0}$: There is not correlation between bug seeding ratio and number of commits.

This hypothesis studies whether there is a relationship between the experience measured in number of commits and the bug seeding ratio. In previous studies [Eyolfson *et al.*, 2011] a direct relationship has been observed between these two variables, although the authors claim that further studies should be done.

Figure 4.21 shows these values where the R^2 has been obtained to observe how good is the fitness of the regression line. As a filter, the less productive developers (less than 20 commits) were removed from the analysis. Each of the dots represents a developer whose number of total commits are represented by the Y-axis and the bug seeding ratio represented by the X-axis. In addition, the regression line is drawn and added its value on the chart.

As can be seen, the regression p-value is not pretty low, except in the case of the

Mozilla Central project. This makes impossible to reject the hypothesis: *There is not correlation between bug seeding ratio and number of commits.*

Surprisingly, this result does not confirm the previous results seen in other communities such as the Linux Kernel or PostgreSQL [Eyolfson *et al.*, 2011]. Indeed, to confirm the results of Eyolfson, the higher a developer is found in the chart, the lefter this should be found.

However, it is also interesting to observe how the developers with a higher expertise (those that are found at the top of each of the charts) usually tend to be found in a vertical line. As an example, the Mozilla Central developers tend to be found between a value of 0.5 and 0.7, while the base of the developers are spread between 0.3 and 0.9. This would be also the case of the Mobile Browser project, where the five most active developers are found between 0.5 and 0.7. Besides, as it has been observed in figure 4.18, the evolution of the developers seem to be closer to the general evolution of the bug seeding ratio than evolving by themselves. Or in other words: the evolution of the bug seeding ratio for a given developers does not increase or decrease once more experience is obtained. Other external factors should be studied.

Hypotheses testing $H_{0,1}$: There is not correlation between bug seeding ratio and territoriality

The definition of territoriality is based on the number of "touched" files by only one developer. However, there is a continuous evolution of the project (except if they are abandoned) what provokes that, even when a developer is working alone in specific areas of the source code, that the probability to find a change from other authors increase.

In addition, the use of Mercurial, as the studied SCM, add another variable to the discussion and this is that the gate-keeper effect partially disappears. In first place, with the existence of the distributed SCM, it helps to easily create branches that later can be merged (as done for instance in Git). Thus, if Git is mined, this will be provide the information that the final committer is the current "gate-keeper" and the information from the actual author is also added.

In the case of Mercurial, it works a bit different: when two branches are merged by a developer, the committer information does not appear (at least if third party applications are not installed in the SCM), what basically removes the gate-keeper information.

In previous studies of territoriality, those have been analyzed using or SVN or CVS. Thus there is a final committer who is submitting the changes to the source code. This, as thought, varies the final results making hard to compare different results when communities use different SCMs, even when both are distributed or centralized.

Regarding to the results, the number, as expected, of files where only one developer has touched during its whole life is always less than a 5%, if the whole life of the project (around 4 o 5 years in this case study) is studied. If the life of the project is divided by

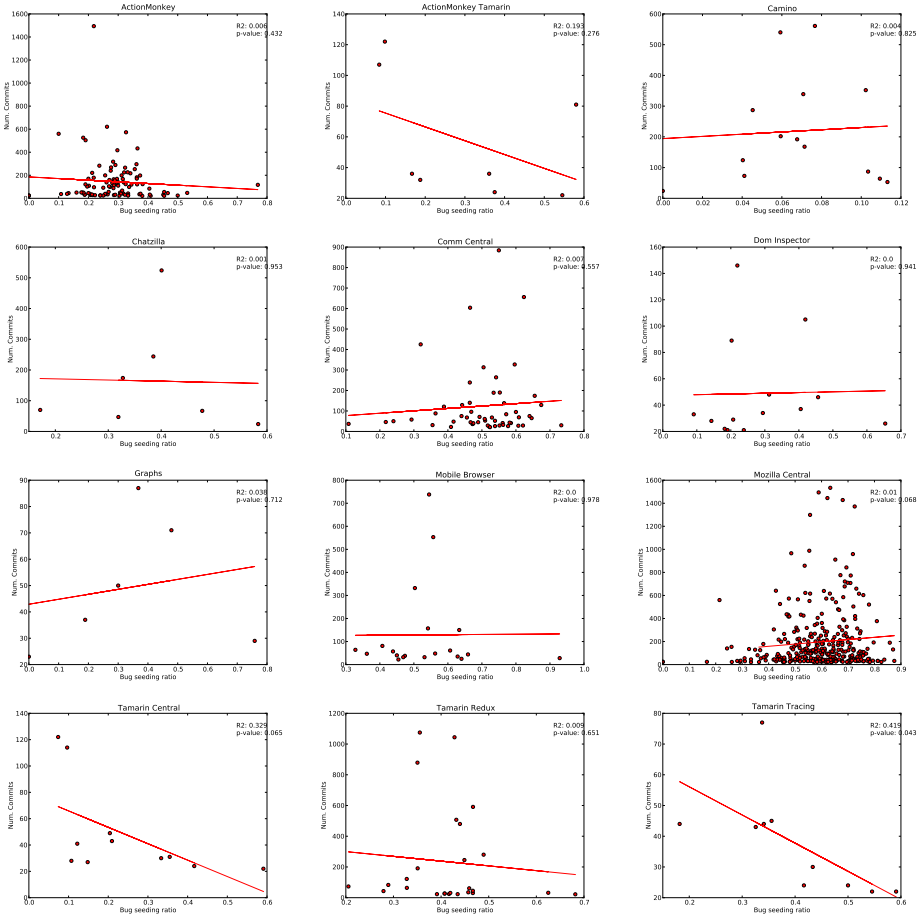

Figure 4.21: *Percentage of bug-seeding commits per developer and aggregated by project*

year, the territoriality of the people, measured in number of owned files, increase.

The analysis shows the comparison between the territoriality values of developers (ignoring those with a territoriality of 0) and the other two metrics used to measure experience: the bug seeding ratio and the number of commits.

Figure 4.22 shows the results, in an aggregated way of the whole set of developers in the Mozilla community with a value greater than 0 for their territoriality and the bug seeding ratio of each of them. As seen, the null hypothesis can be rejected with a confidence of a 99.94%. However the R^2 does not imply a lineal correlation between the two variables.

Hypotheses testing $H_{0,2}$: There is not correlation between bug seeding ratio and fixing activity

The fixing activity has been studied as a potential metric for measuring experience [Ahsan *et al.*, 2010]. This hypothesis testing aims to study the relationship between the fixing activity (expertise) and the bug seeding ratio. From an intuitive point of view, it is expected to find lower ratios of bug seeding activity for those developers that present a higher ratio of fixing activities.

Figure 4.22: *Comparison between territoriality and bug seeding ratio*

For this purpose, figure 4.23 shows this relationship for all of the cases of study. Those projects with a low number of dots in the charts have been removed. As seen in the charts, the p-values are usually low and the null hypothesis can be rejected. However there are others where the null hypothesis can not be rejected, as in the Mobile Browser project. Regarding to the value of R^2, those values are pretty low. This indicates a low level of lineal correlation between the two studied variables.

Hypotheses testing $H_{0,3}$: There is not correlation between number of commits and territoriality

This hypothesis aims to test the relationship between the typical activity carried out by developers and their territoriality coefficient. Since the data available from the territoriality is pretty low (only few developers have the total ownership of one file), the data have been aggregated again.

Figure 4.24 shows the results, where it can be observed that there is not lineal relationship between the two variables. In fact, it is observed a big spread of the dataset where people with a high number of commits show a pretty low level of territoriality. On the other hand, there are people with a high level of territoriality (top of the chart) with a not so high level of activity. In fact the null hypothesis can be rejected with a confidence of 99.99%.

This can be explained from two different perspectives and would create two different clusters of developers. In first place, people that usually work where other developers work. And in second place, people that usually work in isolated parts of the source code.

Those developers with a higher number of commits but a pretty low territoriality would be part of the first set of developers, while people with a high level of territoriality and some activity would be part of the second set of developers.

All in all, the hypothesis $H_{0,3}$ can be rejected.

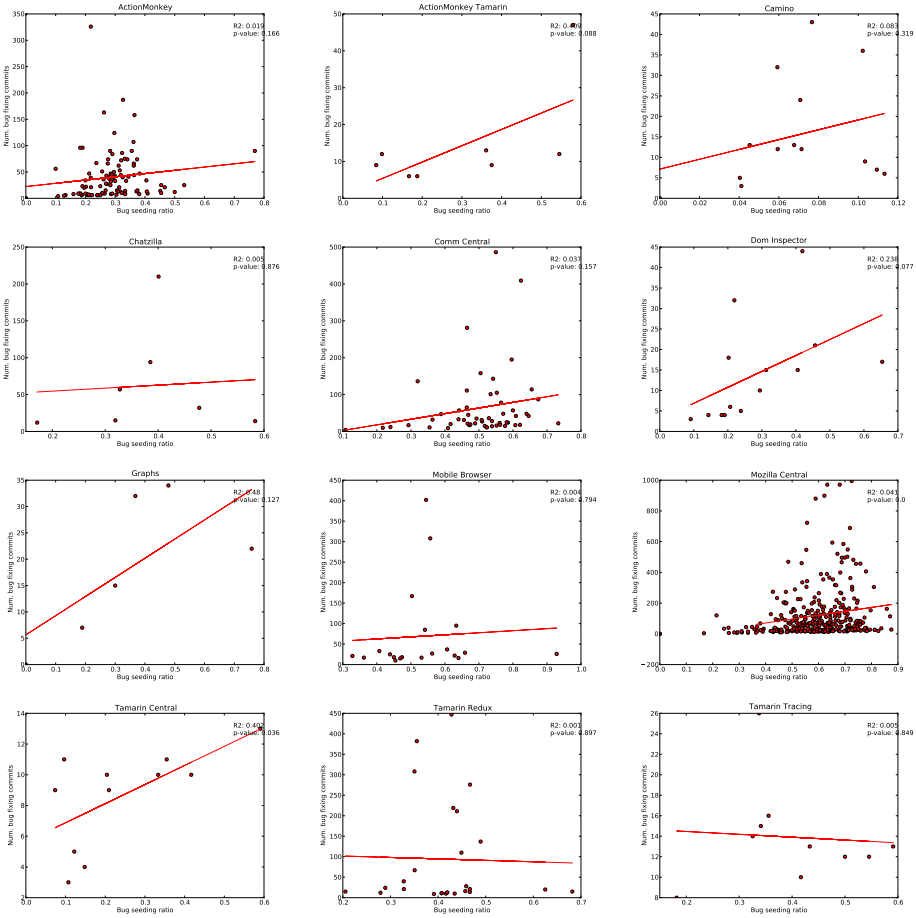

Figure 4.23: *Correlation of bug fixing activity and bug seeding ratio*

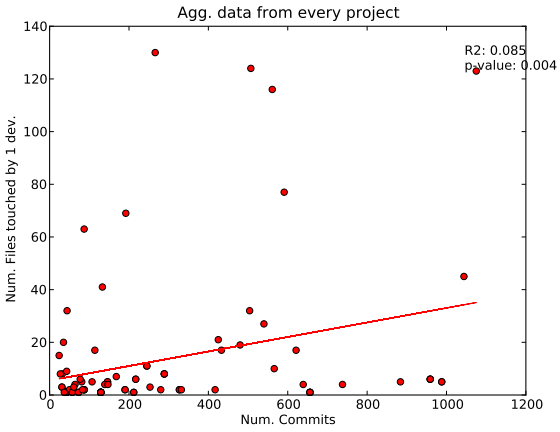

Figure 4.24: *Study of the correlation between the typical activity and territoriality in the Mozilla community.*

Hypotheses testing $H_{0,4}$: There is not correlation between number of commits and fixing activity

This hypothesis aims to study the relationship between the typical activity carried out by developers and their fixing activity. Depending on the role played in a community, different developers should show different types of activity: close to pure maintainers, pure developers, translators or others.

By definition, the fixing activity is a subset of the typical activity. The difference is based on the type of log message left by a developer in the SCM when changing the source code.

Figure 4.25 shows this relationship per project (those with a low level of dots in the chart have been removed).

The results, based on the p-value indicates that the null hypothesis can be rejected. In addition, the results show how a high lineal relationship is found between those developers that are highly active (with a quite high number of commits) and their fixing activity. Thus, these results show that the higher the activity of a developer in a community, the higher her fixing activity.

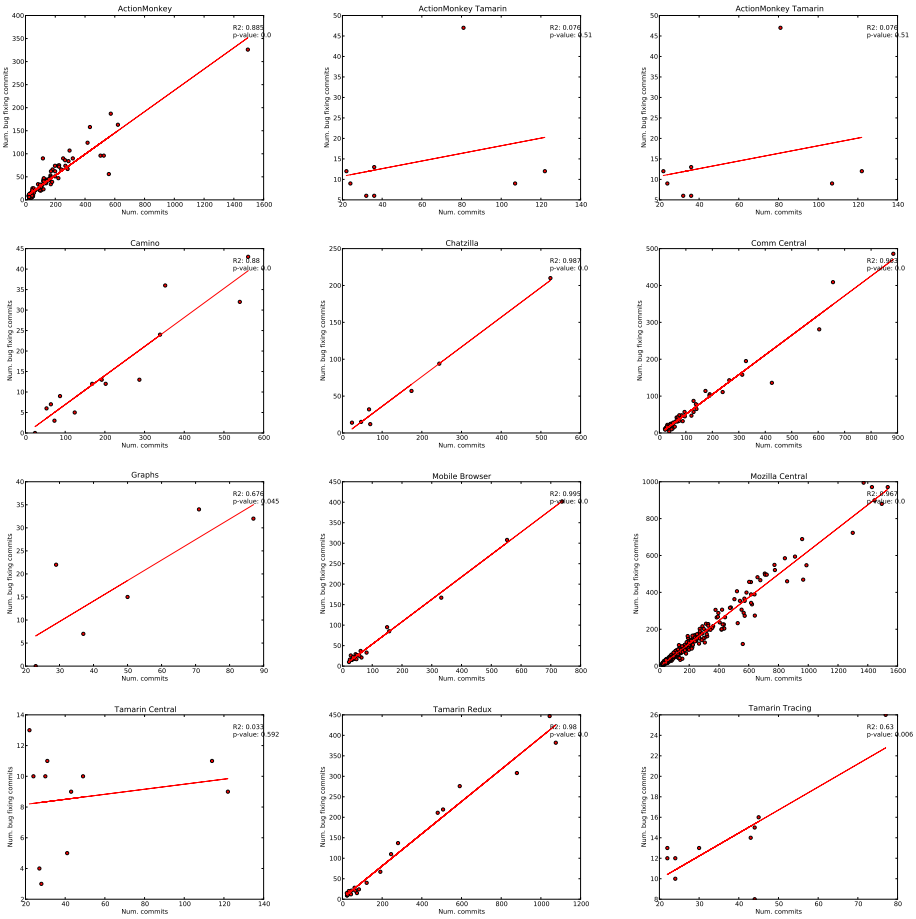

Figure 4.25: *Study of the correlation between typical activity and fixing activity*

Hypotheses testing $H_{0,5}$: There is not correlation between territoriality and

fixing activity

This hypothesis aims to measure if there is any relationship between how territorial is a developer and her typical fixing activity in the source code. The p-value indicates that the null hypothesis can not be rejected. In addition there has not been found lineal relationship between the two studied variables.

Figure 4.26 shows that the null hypothesis can be rejected.

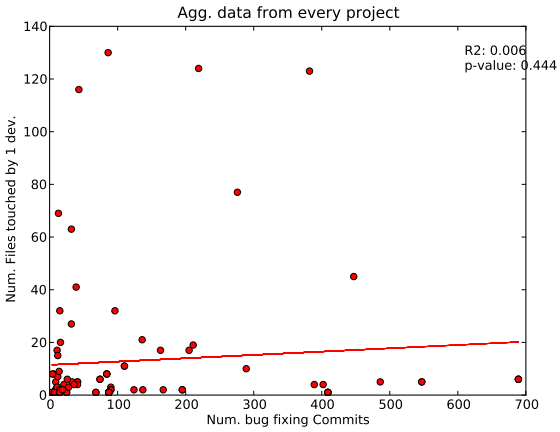

Figure 4.26: *Study of the correlation between territoriality and fixing activity*

4.6.3 Characterization of Developers based on the Seeding and Fixing Activity

This subsection aims to study the differences among developers in all of the cases of study. As a first approach, it has been seen how there are some developers whose ratio of bug-fixing commits is higher or much higher than the ratio of bug-seeding commits. On the other side, others were found to have a lower bug-seeding ratio if compared to the bug-fixing ratio.

Since, most of the specific data presented at the granularity of developers were based on core committer from each of the cases of study, it does not make sense to think that probably a higher expertise means a lower bug-seeding ratio.

Aggregated data of activity, bug-fixing commits and bug-seeding commits

This subsection aims to study the most important developers aggregating the data of commits, bug-seeding commits and bug-fixing commits. This will help us to better understand the evolution of the several developers (in this case the most important ones from the most important communities).

Figure 4.27 shows the aggregated data per developer. The blue line represents the

activity measured in commits, the red line is the number of bug-fixing commits and the green line is the number of bug-seeding commits.

In a first approach, we can see how the previously analyzed activity during the second stage of their "career" in the project in figure 4.27 is now visualized as an increase in the slope of the "line" found for the common activity. In some cases, such as in the top right developer ("jrudeman") there is a real jump, what tell us that in a given moment there was a huge submission of commits to the repository (result also seen in the time series analysis).

Another interesting point is the differences in terms of commits among developers at the level of bug-fixing commits. In some cases the red line is quite close to the slope of the green line (bug-seeding commits). Or in other words, the evolution in the number of "buggy" changes and fixing changes is quite similar. However, other developers do not show this distribution of effort. This could be discussed as a matter of no indication of this data in the message log or, that there are developers that usually tend to fix bugs and others that usually tend to develop and introduce bugs. Thus, there is a difference in terms of maintenance activity depending on the project and on the developers.

This could raise another question related to the different roles that may appear in the FLOSS communities. Are the people with a high rate of bug-fixing commits more important in the maintenance process?. Indeed, are those maintainers?

As an example, taking the two first developers of the charts, ("reed" and "jrudeman") the first one is carrying out more maintenance activities (the red line (bug-fixing commits) shows a bigger slope than the green line (bug-seeding commits)). However, the second chart shows hardly the red line what means that there is not bug-fixing activity detected for this developer.

Evolution of bug-seeding ratio per developer

The evolution of the bug seeding ratio may be altered by other factors different than the developers idiosincrasy itself. Those have been summarized into two main groups.Those seen as external factors, coming from the project idiosyncrasy:

- **Policy defined by the community**: Communities evolve as the software evolves. Those communities which tend to get mature also tend to increase the level of internal quality assurance. In the proposed methodology the quality assurance may affect the commit log messages. In fact, several FLOSS communities did not have any standardized way of specifying the different types of commits. In addition, each community could potentially define any kind of policy what complicates the detection of those fixing commits.

- **Stage of the project**: the stage of the project is also a key factor when carrying out

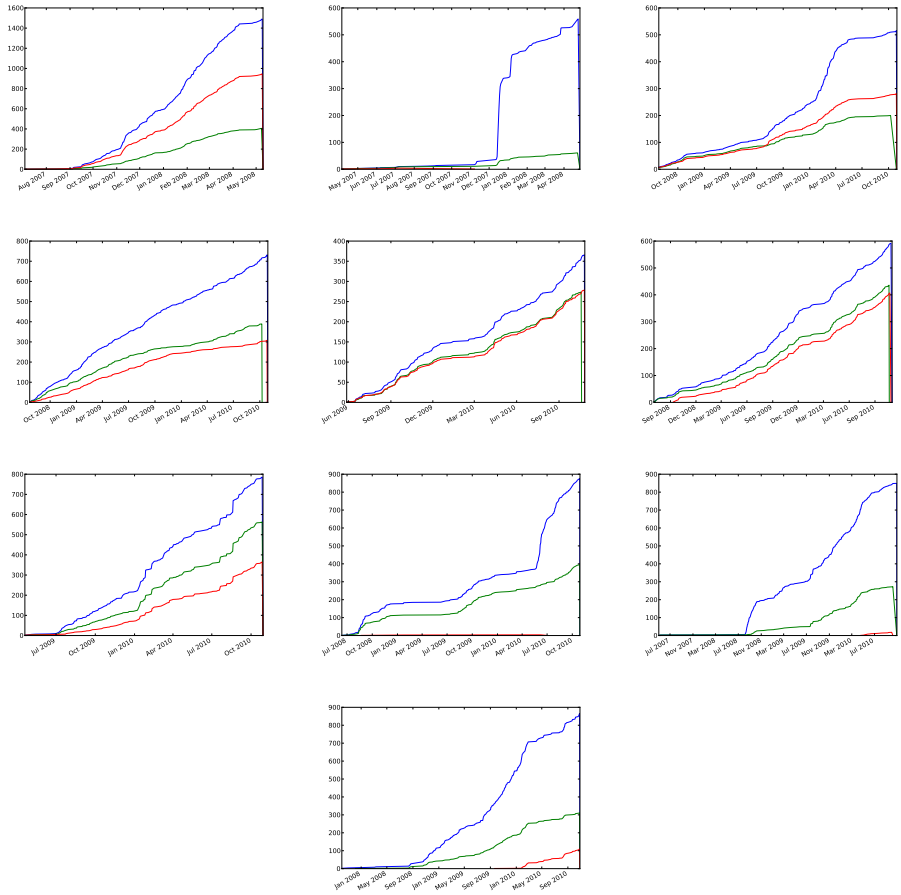

Figure 4.27: *Example of aggregated activity. The two main developers (measured in activity) from each of the five projects analyzed are detailed (ActionMonkey, Comm Central, Mobile Browser, Mozilla Central and Tamarin Redux). The blue line represents the total aggregated activity, the red line the total aggregated bug fixing activity and the green line the total bug seeding activity.*

this type of analysis. Those communities in a very early stage in their development process will tend to find more "buggy" changes than others. They are still in a continuous development process. However those communities in a more mature stage, will tend to focus on maintenance activities in most of the cases. In the first case the addition of lines will be higher, but also and probably there will be an increase in the detected bug-seeding commits.

And those seen as personal factors, coming from the very developer:

- Each developer is **free to specify** or not the different issues that appear in the source code, the bug fixing actions and others. In fact, these are good practices when developing software, however this is not always done. This factor is not measurable since mostly depends on the accuracy of the developers and a hugely and largely

study should be done in these terms. Thus, an increase or a decrease in the ratio of bug-seeding commits will also depend on this circumstances.

All in all, those aforementioned factors may alter the final results of the presented charts in figure 4.27. However if there is a radical change in some of those, this will appear in the charts as a big jump increasing or decreasing the ratio.

By definition, the bug-seeding ratio is defined as the number of bug-seeded commits out of the number of commits and measured in an aggregated way. Thus, what we have is a list of commits and bug-seeding commits with a specific date for each of them. In some cases, a commit could be identified as a normal commit, but also as a commit which introduced an error. Even more, a commit could be identified as a normal commit, a bug-seeding commit and a bug-fixing commit. (A commit fixing an issue that later is identified as a commit which introduced an error in the source code).

Thus, the methodology could show strong deviations at the very beginning and at the very end of the chart. Those deviations appear because of that the initial commits could have been identified as bug-seeding commits (but also normal activity), what will provide a percentage higher than 100%. In the last part of the charts, there could appear a strong decrease in the number of bug-seeding ratio since it takes some time for a commit to raise as a "buggy" change, as seen in previous subsections.

Regarding to the results, we can see how, apart from the initial "odd" values, some of the developers tend to decrease the ratio, while others tend to increase the ratio and while others are just simply stabilized in a given ratio after some evolution.

In general terms, the bigger the net difference between the number of commits and the number of bug-seeding commits, the lower the bug-seeding ratio. As seen in previous figures, the developers with sudden jumps tend to show a sudden decrease in the ratio, like in the top right developer. However in most of the cases, after a "noisy" period most of them tend to stabilize, although the resultant slope is slightly positive or negative.

Depending on the project, and depending on their slopes, the developers tend to show a similar pattern of bug-seeding ratio. This is the case of the last two developers from the tamarin_redux project (Action Script) where the bug-seeding ratio seen in figure 4.18 at the bottom left shows a similar pattern of its two main developers during the last year.

On the other hand, we can find a set of developers whose ratio of bug-seeding commits is decreasing after some months of activity. More specifically, the four first developers from the JS Engine project (actionmonkey) and the Thunderbird project (comm_central) present this pattern. Besides, this decreasing pattern is shown in those projects, but with a bit higher ratio of bug-seeding rate if compared to this developers.

Finally, for the four developers in the middle (mobile_browser - FENNEC and mozill_central - Firefox) these developers show a slope pretty close to 0, what means

that there is a real stabilization of the bug-seeding ratio. This can be seen again in the aggregated charts for the hosted projects.

Thus, even when a majority of the most important developers tend to show a decrease in the bug-seeding ratio, this seems to be totally dependent on the project's evolution and not in the experience showed by each of them. In some cases, this seems to be a key factor, like in the second developer (top right), but not in all of the cases.

Characterizing developers

The results have been previously filtered to study those developers with more than 20 commits.

Figure 4.28 shows these differences in a boxplot chart. That difference is measured as the differences between the two areas represented by the line obtained from the aggregated values of the number of bug fixing and bug seeding commits. Thus, those values less than 0 are those developers whose area of introduction of errors is bigger than the area of maintenance activity. The union of both values would represent the total number of commits done (in some cases, a commit is not included neither in a bug seeding commit nor a bug fixing commit).

The figure at the top, represents all of the differences found and drawn in a boxplot. In this case, most of the developers are found around the 0 value, while the maximum value is close to 50,000 and the minimum value is close to -150,000. This means that the area of those people who usually maintain less than their activity introducing errors is bigger than those who usually maintain more than introducing errors.

This allows to have a first characterization based on the type of the final activity measured in seeding and fixing commits (other types of commits are not taken into account).

In addition, the figure (at the bottom) shows that the median is pretty close to 0.

Regarding to the density of people, since the right part of the bottom figure, or top part of the top figure are smaller in the measured area, this implies a bigger density of people found in that sections.

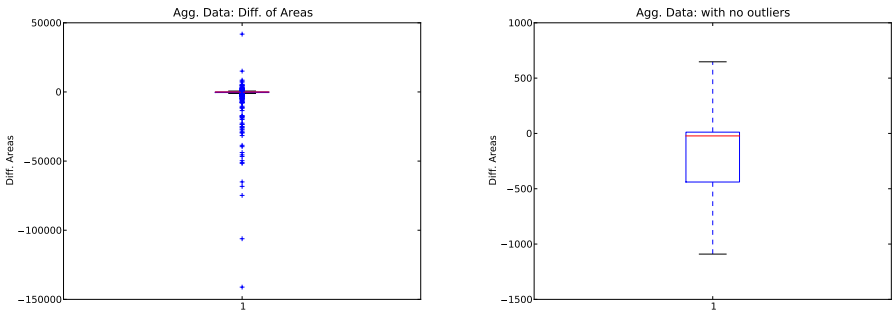

Figure 4.28: *Difference in the area of the developers studied. The more positive a value is, the closer to be a pure maintainer. On the other hand, the more negative a value is, the closer to be a developer who only adds source code but does not maintain. The figure at the left represents the whole dataset, while the figure at the right only depicts the central 50% of those values.*

4.7 Patterns of Activity in a 24 Hours Framework

Up to this section, the concept of bug seeding has been introduced along this dissertation to study the bug life cycle from several perspectives. In addition, the second set of experiments have aimed to study the relationship between experience -quantified by the number of total commits done, fixing activity or territoriality- and bug seeding ratio -quantified by the number of buggy changes out of the total number of modifications to the source code-.

This section aims to study more variables that may affect the quality of the source code when committing changes. In overall, this section is focused on the study of the changes of the developers along a 24 hours timeframe of activity (a typical day).

This study is made by means of an aggregation of the data obtained from the SCM[11]
Based on this 24 hours framework, specific concepts are introduced:

1. In first place, the **distribution of the changes** along a 24 hours framework is studied. This has provoked a filter of the initial dataset leaving only five of the biggest communities: ActionMonkey, Comm Central, Mobile Browser, Mozilla Central and Tamarin Redux. This study is developed in subsection 4.7.1.

2. Secondly, the **time of the day** where there are more chances to introduce an issue is studied. From an intuitive point of view, "odd" hours where the time of work is not usual (e.g.: work carried out after midnight) may be more prone to be buggy than others. This experiment is developed in subsection 4.7.2.

3. Thirdly, the concept of **experience** is brought in context to be compared to the timeframe of activity. A comparison between the traditional definition of "core" developers against the rest of them is done. (subsection 4.7.3).

4. Finally, an observational study is done to introduce the concept of **"comfort" time of work**. This is defined as the typical hours when a developer tend to make changes. However, are the hours out of the typical "comfort" hours more prone to be buggy?. This question is depicted in subsection 4.7.4.

4.7.1 Activity of Developers in a 24 Hours Framework

The aggregation of effort in a 24 hours framework allows to understand how the activity is being carried out. It has been observed two main peaks of activity in general. The first

[11]Taking advantage of the local time found in each of the developers repository. This is possible thanks to the existence of distributed SCM where the time of commit is given by the developer (including timezone). On the other hand, the centralized repositories are found, where the time of commit is always provided by the central server. These experiments are not reproducible in communities or projects using a centralized SCM.

one between 09:00 and 18:00, and the second one between 19:00 to midnight. This shows a double *hump* of activity that divides the day into two main parts. Figure 4.29 shows these two *humps* that define in most of the cases the general activity.

These *humps* of activity help to understand how the effort is distributed in the case of study where there is a *hybrid* type of community. Part of the developers are being paid by companies and others are volunteers. In any of the cases, the highest peaks of activity are found during the typical working hours: let us use *office* timeframe to delimit the time of work between 09:00 and 19:00 where most of the effort is achieved.

Those can also be seen as probability distributions of effort where the time to find a change in the source code is higher during the *office* timeframe than during the rest of the day.

In addition, the bug seeding activity has been aggregated in the same timeslots. This type of activity also follows a double *hump* and follows (visually speaking) a similar distribution if compared to the general one.

This section aims to study the distribution of the resultant aggregation of the data in a 24 hours framework to understand how the effort is being carried out. In general the following research questions are defined:

1. For a given project, is the **annual activity** similar through the years?: this will help to study the distribution of effort once a project is evolving. Depending on the current stage of the project (development or maintenance for instance) the annual activity will tend to differ from the rest of them.

2. Are the **distributions of activity** similar among projects?: this question will help to compare distributions of effort among projects and check if it is possible to aggregate all of the data. Or on the contrary, if the studies must be done project by project. Similar distributions would simplify the size of the total experiments.

3. Are the **distributions of general and bug seeding activity** similar?: As seen in figure 4.29, the distribution of bug seeding and general activity are visually similar. However, it is still needed to confirm this by means of a statistical approach.

Previous research questions can be summarized in the following Null Hypotheses found in table 4.34.

Hypotheses testing $H_{0,0}$: the annual activities of each project is similar. The disaggregation of data per year is specified for the five studied projects and per year of analyzed history: Actionmonkey (table 4.35), Comm Central (table 4.36), Mobile Browser (table 4.37), Mozilla Central (table 4.38) and Tamarin Redux (table 4.39).

As seen in the analysis, depending on the project and the year, the null hypotesis $H_{0,0}$ can be rejected. However, it has been observed, that there are specific years where the

ID	Null hypothesis	Test applied	Attribute
$H_{0,0}$	The annual activity of each project is similar	Kolmogorov-Smirnov	source code commits
$H_{0,1}$	The overall activities in the Mozilla are normally distributed	Anderson and D'Agostino	source code commits
$H_{0,2}$	The overall activities in the Mozilla projects are similar	Kolmogorov-Smirnov (two-tail & directional)	overall commit activity
$H_{0,3}$	The bug-seeding activities in the Mozilla projects are similar	(same)	bug seeding commits
$H_{0,4}$	The bug-seeding and overall activities in the Mozilla projects are similar	(same)	bug seeding and overall commits

Table 4.34: *Null Hypotheses: Effort Distribution.*

Year 1	Year 2	p-value
2007	2008	2.89-e03

Table 4.35: *ActionMonkey Project: Comparison of annual distributions of effort.*

Year 1	Year 2	p-value
2008	2009	1.79-e05
2008	2010	9.31-e04
2008	2011	9.31-e04
2009	2010	9.31-e04
2009	2011	8.18-e07
2010	2011	4.01-e06

Table 4.36: *Comm Central Project: Comparison of annual distributions of effort.*

Year 1	Year 2	p-value
2008	2009	7.32-e05
2008	2010	1.79-e05
2008	2011	**0.62**
2009	2010	**0.38**
2009	2011	2.89-e03
2010	2011	2.73-e04

Table 4.37: *Mobile Browser Project: Comparison of annual distributions of effort.*

Year 1	Year 2	p-value
2007	2008	8.23-e03
2007	2009	0.05
2007	2010	2.73-e04
2007	2011	**0.21**
2008	2009	**0.38**
2008	2010	0.02
2008	2011	0.02
2009	2010	**0.11**
2009	2011	0.02
2010	2011	2.73-e04

Table 4.38: *Mozilla Central Project: Comparison of annual distributions of effort.*

Year 1	Year 2	p-value
2006	2007	9.31-e04
2006	2008	4.01-e06
2006	2009	2.59-e08
2006	2010	1.52-e07
2006	2011	1.79-e05
2007	2008	8.23-e03
2007	2009	7.32-e05
2007	2010	1.79-e05
2007	2011	0.02
2008	2009	0.05
2008	2010	**0.11**
2008	2011	**0.38**
2009	2010	**0.98**
2009	2011	0.02
2010	2011	0.02

Table 4.39: *Tamarin Redux Project: Comparison of annual distributions of effort.*

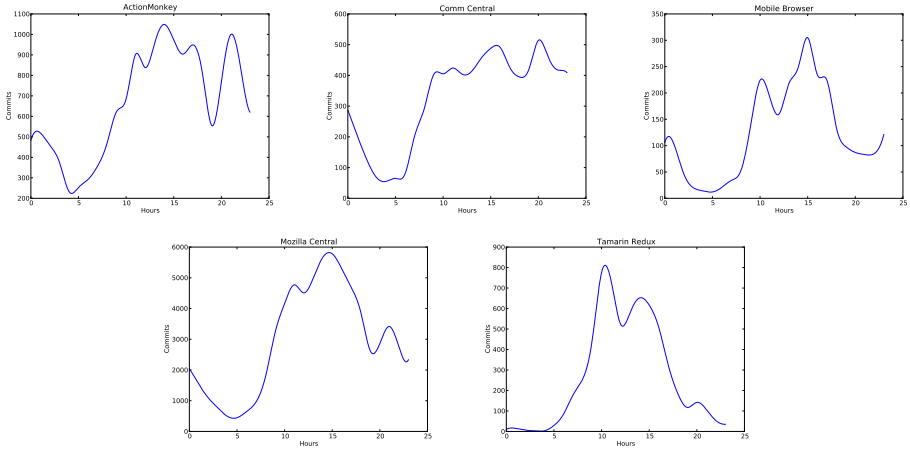

Figure 4.29: *Total activity measured in number of commits in a 24 hours timeframe*

aggregated activity seems to be similar.

- **Actionmonkey**: the years calculated only represents the 2007 and 2008. In this case the hypothesis can be rejected.
- **Comm Central**: the timeframe calculated goes from 2008 to 2011. In this case all of the years are following a different distribution. Thus, the null hypothesis can be rejected.
- **Mobile Browser**: the time of study goes from 2008 to 2011. Although there is not statistical significance to accept the null hypothesis since the p-values are still too low (0.62 between 2008 and 2011; and 0.38 between 2009 and 2010), the null hypothesis can not be rejected in this case.
- **Mozilla Central**: this project shows similar behaviour than the Mobile Browser one. The p-values between 2007-2011, 2008-2009 and 2009-2011 implies that the null hypothesis can not be rejected. In addition, depending on the year, the p-value raises up to 0.2 or 0.5.
- **Tamarin Redux**: this project shows a closer distribution among the years 2008-2011. Indeed, the closest distributions are found between the 2008 and 2011 years with a p-value of 0.98, what implies a similar distribution of activity along the 24 hours framework. Thus the null hypothesis can not be rejected.

The study of the distributions per year allow to identify how similar are the distributions per year and per project. Some years are close to other years what facilitates the aggregation of the whole dataset in order to avoid more complexity when comparing results. In any case, this annual division help us to understand how projects

Project	Anderson Test	D'Agostino Test (p-value)
Actionmonkey	0.5	0.05
Comm Central	1.65	0.11
Mobile Browser	0.5	0.35
Mozilla Central	0.41	0.06
Tamarin Redux	1.32	0.09

Table 4.40: *Testing normality of distributions when aggregating data in a 24 hours timeframe. For the Anderson test, the Anderson-Darling test statistic is provided. If the value is larger than 0.972 for any of the analyzed projects, then the null hypothesis can be rejected. For the D'Agostino normality test, the p-value is provided.*

evolve. However, in the following the history will be aggregated in order to simplify the methodology. Specific divisions will be left as further work.

Hypotheses testing $H_{0,1}$: The overall activities in the Mozilla are normally distributed. This hypothesis study how the distributions of effort along the 24 hours follow a normal distribution. Table 4.40 shows the results when calculating the normality of the distributions. The statistical methods used were the Anderson and the D'Agostino Tests.

Values over 0.972 in the Anderson coefficient test implies the rejection of the null hypothesis. In all of the cases, the critical values are the next array of values: $array[0.513, 0.584, 0.701, 0.817, 0.972]$ for the following significant levels specified for a normal or exponential distribution: $array[15\%, 10\%, 5\%, 2.5\%, 1\%]$. With this approach, the distribution of Comm Central and Tamarin Redux can be said that are not following a normal distribution.

In the case of the D'Agostino normality test, the provided values are the p-value. The results show a different set of results. With a 90% of probability the null hypothesis for Actionmonkey, Mozilla Central and Tamarin Redux can be rejected. However, in the case of Comm Central and Mobile Browser the hypothesis can not be rejected.

Thus, summarizing: the Tamarin Redux project is the only one where both methods show an approximate result of reject the null hypothesis. While in the case of the Mobile Browser both methods can not reject the null hypothesis. Regarding to the rest of the projects, the results show a mix of values that makes not possible to identify the normality of the functions.

Hypotheses testing $H_{0,2}$: The overall activities in the Mozilla projects are similar. This hypothesis aims to study the similarities among the distributions of the analyzed projects. Table 4.41 shows the results when comparing the distributions using a Kolmogorov-Smirnov statistical test. In all of the cases with a probability of 94% the null hypothesis can be rejected.

Hence the distributions aggregated do not follow in any case the same distribution as other projects. This does not allow to aggregate all of the dataset.

Project 1	Project 2	p-value
Actionmonkey	Comm Central	1.79-e05
Actionmonkey	Mobile Browser	5.73-e10
Actionmonkey	Mozilla Central	1.52-e07
Actionmonkety	Tamarin Redux	2.73-e04
Comm Central	Mobile Browser	1.79-e05
Comm Central	Mozilla Central	5.73-e10
Comm Central	Tamarin Redux	**5.05-e02**
Mobile Browser	Mozilla Central	8.80-e12
Mobile Browser	Tamarin Redux	**5.05-e02**
Mozilla Central	Tamarin Redux	2.59-e08

Table 4.41: *Comparison of aggregated distributions among projects.*

Hypotheses testing $H_{0,3}$: The bug-seeding activities in the Mozilla projects are similar. This hypothesis aims to compare the bug seeding commits distributions in a 24 hours framework. These distributions are depicted in figure 4.30. Regarding to the shape of the resultant datasets, the two main peaks of activity that were studied in figure 4.29 are in some cases diluted. This is for instance the case of the Mobile Browser or Tamarin Redux projects, where there are some peaks of activity during the "office" timeframe, but not during the evening timeframe of the day. In other cases, such as the Comm Central project, the second peak of activity is even higher during the evening timeslot what introduces one of the research questions that were detailed at the beginning: are specific part of the day more prone to be *buggy*?. Finally, regarding to the ActionMonkey and Mozilla Central projects, the two peaks of activity are still there, but visually speaking smaller.

With respect to the distribution of the dataset, table 4.42 makes a comparison between the several bug seeding commits distributions in order to study how similar they are. As can be seen in the results, only in the case of the couples Actionmonkey - Comm Central, Comm Central - Tamarin Redux, and Mobile Browser - Tamarin Redux the null hypothesis can not be rejected.

Hypotheses testing $H_{0,4}$: The bug-seeding and overall activities in the Mozilla projects are similar. This hypothesis aims to study similarities between the distribution of bug seeding and overall activity in a 24 hours framework. Table 4.43 shows the results when comparing the bug seeding and overall activity. With a 94% of probability the null hypothesis can be rejected.

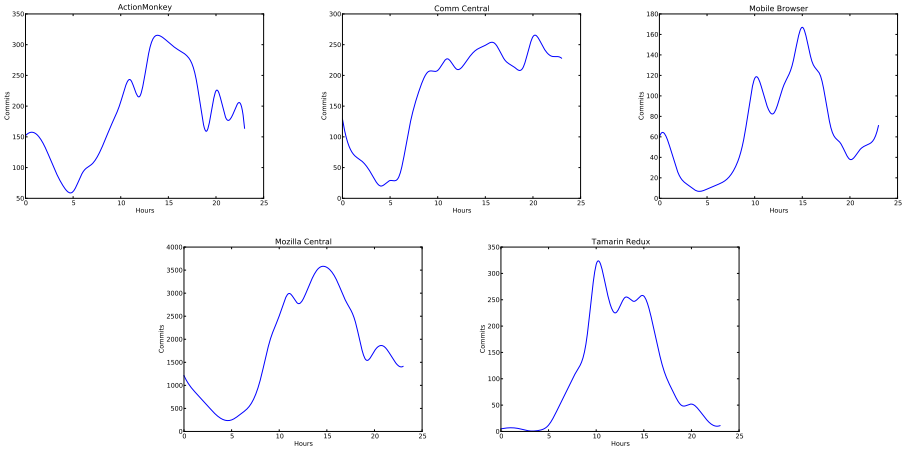

Figure 4.30: *Bug seeding activity measured in number of commits in a 24 hours timeframe*

Project 1	Project 2	p-value
Actionmonkey	Comm Central	**0.62**
Actionmonkey	Mobile Browser	4.01-e06
Actionmonkey	Mozilla Central	5.73-e10
Actionmonkety	Tamarin Redux	2.89-e03
Comm Central	Mobile Browser	1.79-e05
Comm Central	Mozilla Central	7.43-e11
Comm Central	Tamarin Redux	**0.05**
Mobile Browser	Mozilla Central	8.80-e12
Mobile Browser	Tamarin Redux	**0.11**
Mozilla Central	Tamarin Redux	5.73-10

Table 4.42: *Comparison of aggregated bug seeding distributions among projects.*

4.7.2 Study of the Changes in specific Parts of the Day with more buggy Activity

From the management perspective, the cases of study have shown different distributions when comparing the bug seeding and overall activity. In addition, the distributions of activity among projects in the case of the bug seeding and overall activity is also different.

It is also known that there is activity around the clock and it has been observed specific peaks of bug seeding activity during specific parts of the day. This leads to the next research question related to the study of specific parts of the day that may be more buggy than others. Among others, factors may include deadlines that forces developers to work more hours, tiredness, less or more experienced developers and others.

In order to answer the research question, the day has been again divided into 24 hours. The bug seeding ratio, already introduced in section 4.6 is compared hour by hour.

Table 4.44 shows the null hypotheses to be tested.

Figure 4.31 shows the results of this comparison per project. The dashed line represents

Project	K-S comparison
Actionmonkey	4.03-e09
Comm Central	4.01-e06
Mobile Browser	**5.05-e02**
Mozilla Central	**5.05-e02**
Tamarin Redux	**5.05-e02**

Table 4.43: *Comparison of aggregated bug seeding distributions among projects.*

ID	Null hypothesis	Test applied	Attribute
$H_{0,0}$	The bug seeding ratios are similar among projects	Kolmogorov-Smirnov	bug seeding ratio
$H_{0,1}$	The bug seeding ratio is affected by the time of the day	Kolmogorov-Smirnov	bug seeding ratio
$H_{0,2}$	The bug seeding ratio is uniform along the day	Kolmogorov-Smirnov	bug seeding ratio

Table 4.44: *Null Hypotheses: effort distribution of the bug seeding ratio.*

the overall activity (as seen in figure 4.29). The continuous line represents the bug seeding ratio during the 24 hours timeframe. In addition the maximum and minimum bug seeding ratio values are represented by the two horizontal lines.

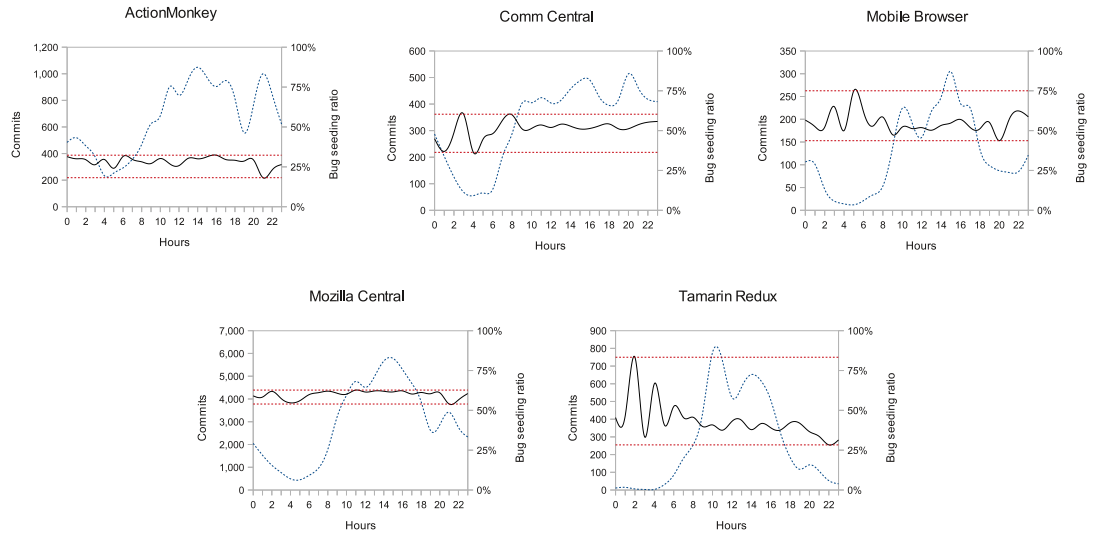

Figure 4.31: *Total activity (dashed) and bug-seeding ratio (continuous) per project*

The visual analysis of the five projects produces three results:

- Firstly, related to the typical hours of work and the maximum bug seeding ratio found. It is clear in three out of the five projects that the highest peak of bug seeding ratio is found into the wee hours aproaching the daybreak (between midnight

and 07:00). Those projects are Comm Central, Mobile Browser and Tamarin Redux. Regarding to the other two projects, the bug seeding ratio is flatter and quite uniform along the day.

However, it is observed that during the wee hours, the activity is pretty low, what may decrease the statistical significance of the results. Bringing in context again the potential causes of bug seeding commits: why are some commits being done during this timeframe?: among others deadlines or people working on specific hours. However this would need an extra qualitative analysis

- Secondly, related to the bug seeding ratio itself. Depending on the project, the average of percentage of bug seeding ratio vary from a 25% of the ActionMonkey project to the 60% found in Mobile Browser. Thus, why do these differences exist among projects?. A potential explanation is a threat to validity since these values are based on the data left by developers in the source code. The more verbose they are, the more bugs are found.

  Regarding to this point, the Mozilla community follows a policy of peer review for any change to the source code in the defined as "core" projects (Mozilla Central, Comm Central and Mobile Browser[12]). Thus, this can also be seen as a factor that keeps stable the bug seeding ratio and helps in how the corrective maintenance is undertaken.

- Thirdly, related to the differences between the maximum and minimum bug seeding ratio. The smallest differences are found in ActionMonkey and Mozilla Central projects, where the values tend to be more uniform along the 24 hours timeframe. In addition, those projects are the two with the highest registered activity reaching peaks of thousand of commits. On the other hand, the Mobile Browser, Comm Central and Tamarin Redux are the ones presenting the lowest activity.

  In addition, the most irregular timeframes (before the daybreak) are usually found in timeframes where the peaks of activity hardly reach a hundred of them. This pretty low activity may bias the final results (although this can also be seen as a potential source of bugs).

  Finally, this could be seen as a result of the differences between young and mature projects, since young projects are still in a developing phase where requirements and source code in general usually change and its continuously modified. Thus, the differences are higher.

[12] `http://www.mozilla.org/hacking/commit-access-policy/` where Mozilla Central is found in the "mozilla-central" directory, Comm Central in the "comm-central" directory and Mobile Browser in the "mobile-browser" directory. All of them can be found at `hg.mozilla.org`. The ActionMonkey also had its own review policy: `http://www.mozilla.org/hacking/reviewers.html`

Hypothesis testing ($H_{0,0}$ – The bug-seeding ratios are similar for all the projects

The bug seeding ratios that are depicted in figure 4.31 were compared between pairs of projects. This experiment aims to determine if they may be considered as similar. A two-tail Wilcoxon test was run between any two pair, and the results (in the form of the resulting p-values) are summarised in table 4.45. For all of the comparisons, except for the Comm Central - Mobile Browser, the null hypothesis can be rejected. Regarding to the Comm Central - Mobile Browser comparison, the null hypothesis can be weakly rejected with a 90% of confidence.

Project 1	Project 2	p-value
ActionMonkey	Comm Central	6.202e-14
ActionMonkey	Mobile Browser	6.202e-14
ActionMonkey	Mozilla Central	6.202e-14
ActionMonkey	Tamarin Redux	7.517e-11
Comm Central	Mobile Browser	*0.08141*
Comm Central	Mozilla Central	7.272e-10
Comm Central	Tamarin Redux	6.417e-05
Mobile Browser	Mozilla Central	2.74-e04
Mobile Browser	Tamarin Redux	4.831e-07
Mozilla Central	Tamarin Redux	5.388e-08

Table 4.45: *Wilcoxon test – Hypothesis $H_{0.0}$*

Hypothesis testing ($H_{0,1}$) – The bug-seeding ratios are affected by the time of the day

The goal of this hypothesis is to test the differences in bug seeding ratio between the several peaks of activity. As seen in figure 4.31, the two main peaks of activity take place during the named "office hours". However, the bug seeding ratio peaks are usually found out of the "office hours". Thus, this experiment aims to check if the time of work out of the main "hump" of activity is more prone to be "buggy".

In order to study these differences, the day has been divided into two main timeframes: the first one includes all of the first hump of activity between 08:00 and 19:00. The second timeslot of study consists of the rest of the hours of the day.

Table 4.46 summarises the outcomes of the tests. Regarding to the comparison of activities, using the visual data it is possible to determine that there are main differences in terms of activity between the two pre-defined slots of activity. While in the "office hours" period of activity, the main peaks of it are found, the rest of the day presents a much lower ratio.

The first column represents the p-values of the null hypothesis testing if the bug seeding ratio is the same in the two timeslots. In this case the values determine that the null

Project	Ratio (two-tail)	Ratio (greater)
ActionMonkey	**0.21**	**0.90**
Comm Central	**0.86**	**0.59**
Mobile Browser	**7.319-e02**	3.65-e02
Mozilla Central	8.25-e03	**0.99**
Tamarin Redux	**0.37**	**0.18**

Table 4.46: *Mann-Whitney test – Hypothesis $H_{0.1}$*

hypothesis can not be rejected. This raises the question of similar bug seeding ratio activities along the day. In addition, the second column compares if the second timeslot (19:00 - 08:00) is greater than the first one (08:00 - 19:00). The results show that this hypothesis again can not be rejected.

Summarizing, it is possible to say that the overall activity along a day is different from a visual point of view. However, deepening in the comparison of distributions in the bug seeding ratio, it is not possible to say that the bug seeding ratio distribution in the first and second timeslot are different. Finally, the result of being greater the distribution of bug seeding ratio during the 19:00 - 08:00 period makes possible to point to the fact that this second period of activity is more prone to present issues. Thus, in order to avoid so high "buggy" activity some recommendations related to monitor and specifically peer review the actions after the main peaks of activity should be implemented. An alert system for developers changing the source code during those timeslots may improve the maintenance effort in the project.

Hypothesis testing ($H_{0,2}$ – The bug seeding ratio is uniformly distributed

This hypothesis aims to study if the studied bug seeding ratios follow an uniform distribution. This would help when understanding how the errors are being introduced in the source code. By definition a discrete probability uniform distribution means that the values found in it are equally distributed and spaced. Thus, regarding to the study of the distribution followed by the bug seeding ratio, this hypothesis studies whether the bug seeding activity (if compared to the typical activity) presents specific patterns, or if this appears to be uniformly distributed in any hour of the day.

Table 4.47 shows the results for the comparison of the distributions. In each of the cases, the distributions have been compared with a randomly created uniform distribution with the lower and upper bounds limited by the maximum and minimum bug seeding ratio in each project.

The first column represents the p-values when comparing the distribution of each project with the randomly created uniform distribution. the null hypothesis can be rejected in the case of the typical distribution with a confidence in all of the cases of a 97%. Regarding to the accumulative distribution, the p-values are pretty high and thus, the null hypothesis can not be rejected.

Project	p-value (typical)	p-value (accumulative)
ActionMonkey	5.05-e02	0.98
Comm Central	5.05-e02	0.99
Mobile Browser	2.73-e04	0.98
Mozilla Central	2.13-e02	1.0
Tamarin Redux	1.79-e05	0.38

Table 4.47: *Testing uniformity of the bug seeding ratio distributions - Hypothesis* $H_{0.2}$

The second column represents the cumulative distribution and the comparison between each couple. The values here presents an opposite case where the null hypothesis for the cumulative distribution can not be rejected.

Regarding to the visual analysis seen in figure 4.31, it is interesting to mention the "a priori" uniformity of the projects ActionMonkey and Mozilla Central. However, in the rest of the projects, there is always some noise at the beginning of the day where the bug seeding ratio has not been stabilized. Indeed, the Tamarin Redux, where the big differences between the maximum and minimum bug seeding ratios have been detected is the one where in table 4.47 shows the worst results when comparing distributions.

4.7.3 Experience and 24 Hours of Activity Timeframe: Core and Non-core Developers

This subsection aims to study the relationship between experience, as donde in section 4.6.2 and the type of activity developed in a 24 hours framework. The concept of experience is again based on the number of commits. In order to simply the current analysis, the developers have been divided between core and non-core developers what provides two main groups of people. The definition of core developer is based on the definition given by Crowston [Crowston & Howison, 2005]: "those developers who usually carry out the 80% of the work".

It is expected to find less "buggy" changes by those developers found in the core group since the more experienced a developer is, the higher the quality of the source code and the lower the bugs involuntary introduced in the source code will be found.

Regarding to the method, the previous figure 4.31, where the continous line represents the bug seeding ratio along 24 hours, is now divided into these two groups of developers.

The hypotheses are summarized in table 4.48.

Hypothesis testing ($H_{0,0}$). The bug-seeding ratios of core and non-core developers are similar.

This hypothesis aims to study the similarities in the bug seeding ratio between the core and non-core developers. A Wilcoxon test is used, using as independent variable the bug seeding ratio. In addition, the evaluation of the ratio was only done during the last

ID	Null hypothesis	Test applied	Attribute
$H_{0,0}$	There are not differences in the bug seeding ratio between core and non-core developers	Wilcoxon test two-tail	bug seeding ratio
$H_{0,1}$	The non-core developers bug seeding ratio is greater than the core developers'	Wilcoxon test one-directional	bug seeding ratio

Table 4.48: *Null Hypotheses: Core and non-core developers*

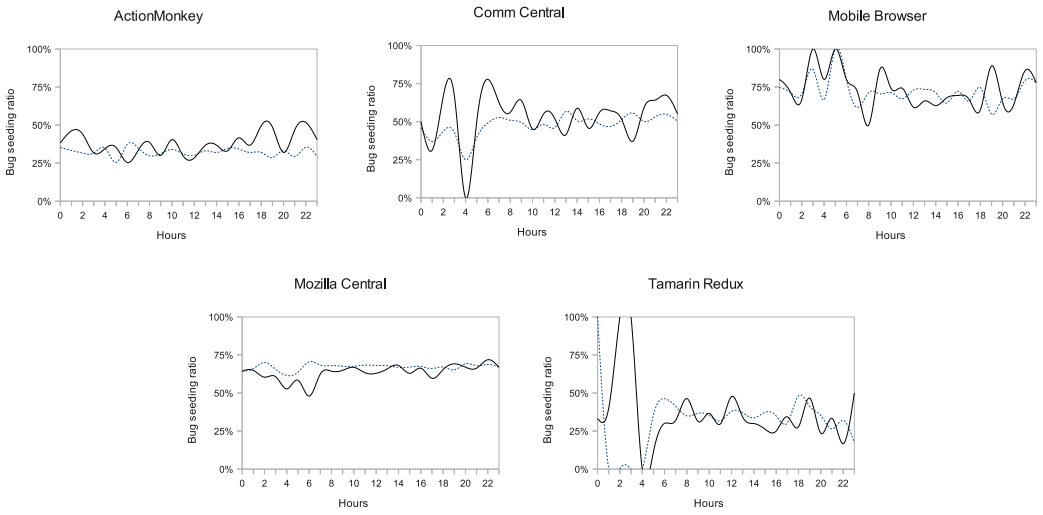

Figure 4.32: *Comparison of bug-seeding ratio introduced by core (dashed) and non-core (continuous) committers.*

year of activity in order to focus on the actual core team of developers. This helps to ignore previous members of the core team that are not currently working.

The results are shown in table 4.49.

Project	Bug-seeding ratio (two-tail)
ActionMonkey	6.76-e03
Comm Central	**1**
Mobile Browser	2.78-e03
Mozilla Central	6.09e-06
Tamarin Redux	**7.43-e02**

Table 4.49: *Unpaired Wilcoxon test – Hypothesis $H_{0.0}$*

The results show with a high confidence that there are not similarities between the distribution of bug seeding ratio between core and non-core developers. Just in one case, the Tamarin Redux project, the rejection of the null hypothesis can be seen as weak (at least with a confidence of a 92%). The Comm Central project shows also a high value for the p-value, however the coefficient

Hypothesis testing ($H_{0,1}$). The bug-seeding ratios of non-core developers is greater than the core developers.

This hypothesis aims to study if the bug seeding distribution in the case of non-core developers is greater than the core developers. Table 4.50 shows the results.

Project	Bug-seeding ratio (one-directional)
ActionMonkey	**0.99** (opposite)
Comm Central	**0.50**
Mobile Browser	**0.99**
Mozilla Central	**1**
Tamarin Redux	**0.96** (opposite)

Table 4.50: *Unpaired Wilcoxon test – Hypothesis $H_{0.1}$*

If the Comm Central project is ignored, for the rest of the projects it is possible to reject the hypothesis that the non-core developers are introducing more relative bugs in the source code. Indeed, in the case of the ActionMonkey and Tamarin Redux projects, the null hypothesis that can not be rejected is that the distribution of bug seeding ratio in the case of core developers is greater than the non-core developers.

4.7.4 Observational Study on the Comfort Timeframes of Activity

This subsection introduces the idea of "comfort time of work". This is defined as the time of the day where developers usually work. As an example, let us assume a typical office work hours: from 08:00 to 17:00 plus one hour to have lunch. In specific cases, such as deadlines, it is possible that employees were "forced" to stay working for a longer time due to schedules or deadlines. The comfort time in this case is where the employee usually works and this is from 08:00 to 17:00 and not the rest of the timeslot.

The goal of this study is to compare the typical timeframe of working hours (defined as comfort time) with the rest of the working hours. It is expected to find a higher bug seeding ratio out of the comfort time.

This observational study is focused on the two main contributors of the five studied projects during the last year of activity (approximately June 2010 - June 2011, except in the case of the ActionMonkey project whose period of activity studied is between 2007 and 2008).

In addition, the sample of study is not representative of the whole population of the core developers of the Mozilla community. Depending on the project, these two developers represents up to a 50% of the total (like in the Mobile Browser project) or just a 6% of the total (like in the Mozilla Central project).

This study will provide specific details of each of the developers (at least the most experienced ones, according to the definition of experience used along this dissertation)

and will help to understand the usual time of work.

Finally, in this experiment, an initial classification of developers is done in order to observe the typical patterns of activity of each of them. Based on the initial results, the patterns found are the following:

1. *Working in a more traditional way*: most of the activity is done between 08:00 and 18:00.

2. *Working during the evening and the wee hours*: most of the activity is carried out between 18:00 and 03:00.

3. *Working through the day*: people whose aggregated pattern of activity do not show any specific timeframe of activity. The activity is being undertaken during the whole 24 hours.

4. *Working in a morning and evening timeframe*: People that usually work during the traditional way, but adding some effort after dinner time (from 19:00 to 00:00).

The following paragraphs show a list of developers that fitted in each of the aforementioned patterns of activity. In addition to the activity developed and aggregated in a 24 hours framework, the bug seeding ratio is visualized.

1- Working in a more traditional way

Figure 4.33 shows the activity pattern (dashed line) and the bug seeding ratio (continuous line) for the Tamarin Redux developer -Steven- and the Comm Central developer -David-. As it can be seen, during the last year their activity fits in the typical "office" hours: between 08:00 and 18:00. The highest peaks of activity are found in both cases at the very beginning of the working hours, while during the rest of the day the aggregated activity decreases.

Regarding to the bug seeding ratio, the highest peaks of activity are detected out of the typical working hours. In the case of Steven, the highest peak of bug seeding ratio is found just before the comfort timeframe. While in the case of David, the peaks of bug seeding ratio are found at the beginning and at the end of the comfort timeframe, but also around midnight. It is specially interesting the last example, where the afternoon hours are also more prone to be buggy.

2- Working during the evening and late night hours

Figure 4.34 shows the activity for those developers detected as being part of the pattern of activity of evening and the wee hours. Both developers studied from the ActionMonkey projects are included in this pattern. As seen in the figure, the typical activity of these developers starts in the afternoon and reach midnight (in the case of Jrudeman) and 04:00 in the case of Reed. The main peaks of activity are found around 22:00 in both cases.

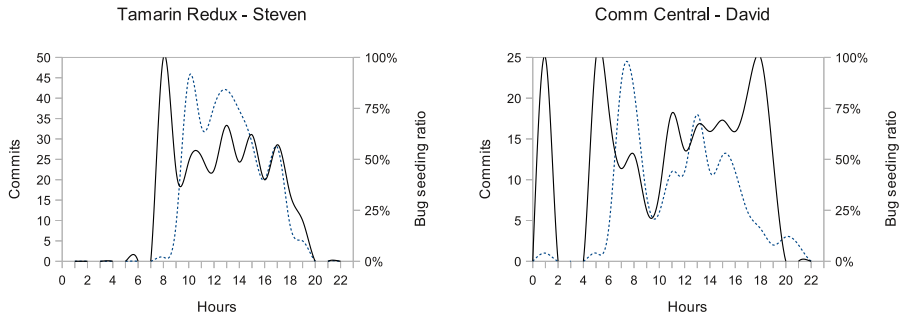

Figure 4.33: *Total activity (dashed) with bug-seeding ratio (continuous) in a 24 hours framework: main developer of the Tamarin Redux and Comm Central*

With respect to the bug seeding ratio, in both cases, the highest peaks of relative buggy changes are found out of the typical working hours. In the case of Jrudeman, those are found at the beginning of the typical timeframe of activity (around 10:00 - 12:00). In the case of Reed, those are found in the wee hours, around 05:00 - 07:00. It can be appreciated how during the rest of the working hours, the bug seeding ratio is quite low if compared to the rest of the core developers. These two developers barely reach a 25% of relative buggy changes.

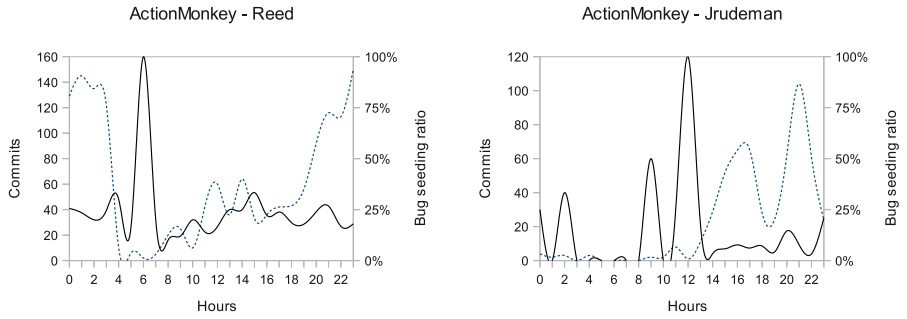

Figure 4.34: *Comparison of total activity (dashed) with bug-seeding ratio (continuous) in a 24 hours framework. Main developers of the ActionMonkey project*

3- Working through the day

Figure 4.35 shows the activity of those developers that do not fit in the rest of the patterns (although it can be claimed that partially some of them may be part of others patterns). At almost any time of the day, activity has been registered. In this set of developers two projects are included: Mozilla Central and Mobile Browser.

It is specially interesting the case of the Mobile Browser developer "Vivien" since there is not a visible pattern of activity. There are several peaks of activity along the day and in all of the hours of the day, activity has been registered. In addition, the bug seeding ratio is also quite similar through the day, even in those hours with the lowest activity:

between 02:00 and 08:00.

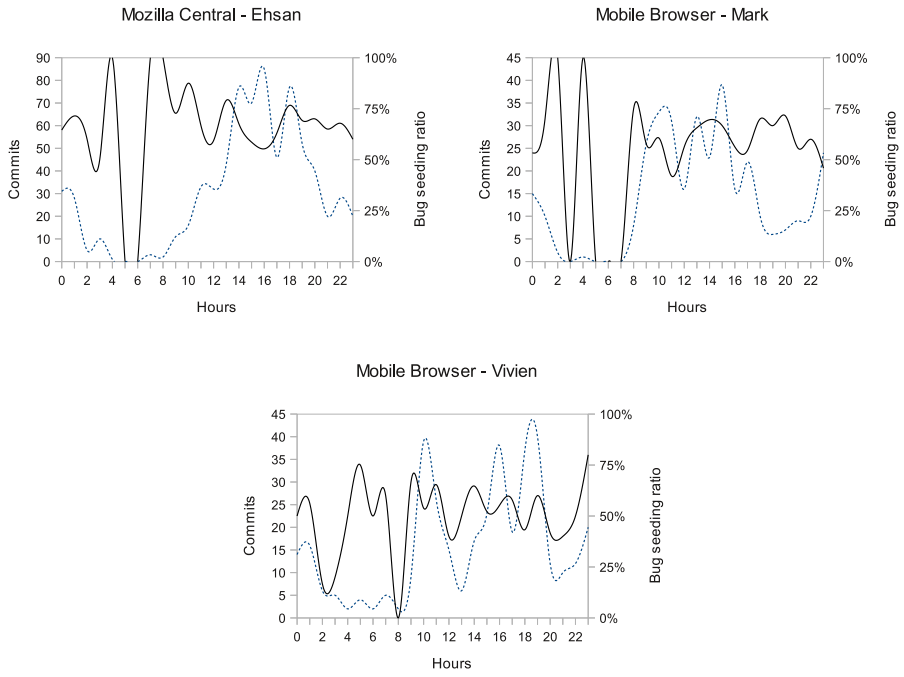

Figure 4.35: *Total activity (dashed) and bug-seeding ratio (continuous) in a 24 hours framework. Top: Mozilla Central developers. Bottom: Mobile Browser developers.*

Regarding to the rest of the other two developers, main peaks of activity are found during the typical working hours in an office, but also after those hours, during the evening and during the wee hours. Particularly interesting the decrease along the day of the bug seeding ratio for the developer "Ehsan" from the Mozilla Central project. While the activity is increasing, the bug seeding ratio decreases, however, in those lowest parts of activity, the peaks of bug seeding ratio are found. This can also been extrapolated to the other developer -Mark-: out of the typical working hours, the bug seeding ratio peaks are found. However in this case, the bug seeding ratio during the typical working hours tend to keep stable.

4- Working in a morning-evening timeframe

Finally, three more developers were found to follow the last patter of activity: working during a typical "office" time, but also during the evening. In all of the cases, activity after 23:00 is hardly found and they do not start untill 06:00 - 07:00.

This pattern of activity also presents peaks of bug seeding ratio at the beginning and at the end of the working hours. It is particularly seen in the case of the developer named as "Lars" and David. The case of Mark, from the Comm Central project does not seem to present too high peaks of bug seeding ratio, although, they exist.

Discussion

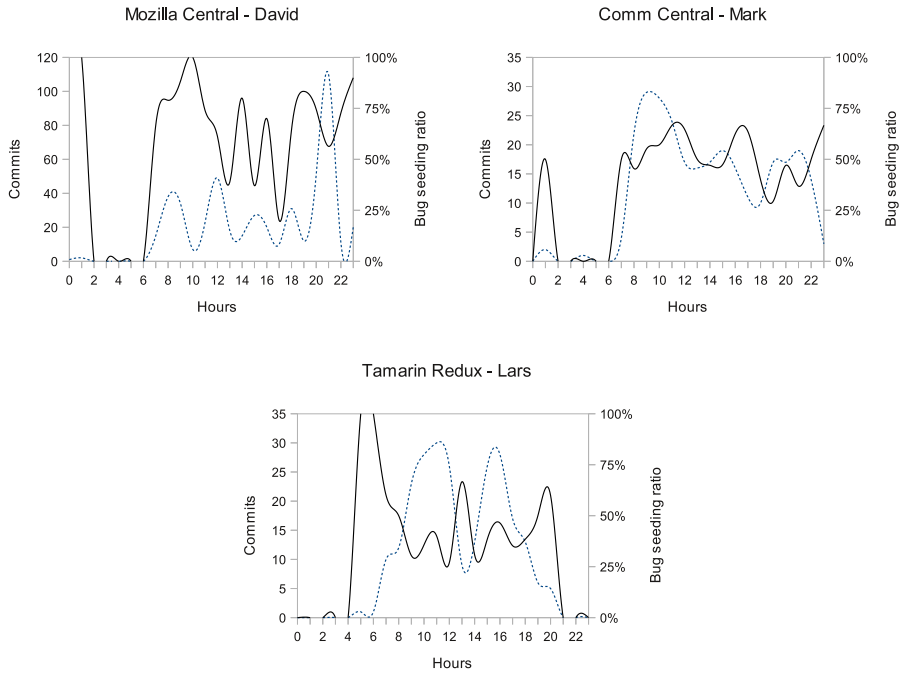

Figure 4.36: *Comparison of total activity with bug-seeding ratio in a 24 hours framework. Left: Comm Central developer. Right: Tamarin Redux developer*

From the results above, it is visible how the activity patterns could be divided in (at least) four main sets: by defining a "comfort area", it could be even possible to detect potential bug-seeding activities based on the activity of single committers. This could be very useful since, based on her past bug-seeding activity, the developer receives personalized information about her specific way of developing. This could be quite easily added to the SCM system in order to give pieces of personalized advice to core and non-core developers, but also to project managers: such reports could help to understand better her developer team and to improve the decisions when adding effort to peer review activities. This is even more important for the Mozilla community where there is a strong policy related to the "core" projects.

Such classification of activity could be further enhanced to highlight the differences between developers working on behalf of their companies, with a well defined comfort timeframe, as the ones in Figure 4.33 or in Figure 4.36, both affiliated to Adobe; and the developers working as volunteers instead, with a less structured or regular way of developing, and working in a 24 hours frame as in Figure 4.35, where developers do not use a specific affiliation, but the generic Mozilla community.

It is also remarkable how different patterns of activity are observed in different communities: the main developers from the ActionMonkey, but also from Mozilla Central and Mobile Browser all follow a similar pattern of activity, related to the community. This

happens also for the Tamarin Redux developers, although one of them presents a small hump during the evening time. This consistency could lead to an extrapolation of the Conway's law [Conway, 1968] where the structure of the source code is a reflection of the hierarchical structure of the people developing it: specific activity patterns could therefore be seen as a reflection of the hierarchical structure, which in turn could drive to specific bug-seeding patterns.

Chapter 5

Results

''Everything should be made as simple as possible, but no simpler''
Albert Einstein

This chapter aims to summarize the main results obtained in chapter 4 and their potential applicability in other environments different from the Mozilla Community.

This chapter is as follows: in first place, section 5.1 aims to list all of the threats to validity found in the study: internal, external and construction validity.

This threats to validity will raise potential questions that are discussed in section 5.2. This discussion section will be divided into several subsections, aiming to address as many as possible points of interest. Among others: methodological or results discussion is addressed.

And finally, section 5.3 will specify the main results found based on the three main research goals: the study of the bug life cycle, human factors that may introduce more errors such as experience and human factors such as the time of the day.

5.1 Threats to Validity

This section will detail the threats to validity that may affect the dataset, tools and results. It is worth stressing the point that this dissertation has basically used an empirical and quantitative approach. From this point of view, several authors have already faced the difficulties of detailing the threats to validity, such as [Runeson & Höst, 2009], [Fernandez-Ramil *et al.*, 2008] or [Kitchenham *et al.*, 2002]. Three main aspects of validity have been identified:

- **Construct validity**: this aspect is focused on the validity of the metrics used in the empirical approach and if they represent what it is expected. In the context of

this thesis, this validity aspect will focus on the several metrics used and potential problems that may bias the result of the metrics when retrieving them.

- **Internal validity**: this aspect is focused on the validity of the tools used. Also, other factors that may alter the results or may affect some of the metrics, even when it is not known their effect should be mentioned. In the context of this dissertation, this validity aspect will focus on the tools and their validation. In addition, other factors that may alter the final value of the metrics should be taken into account.
- **External validity**: finally, this aspect is focused on the extensibility of the research. In the context of this dissertation, this is focused on the potential applicability of the method to other FLOSS communities or proprietary systems.

5.1.1 Construct Validity

This subsection aims to identify the potential threats to validity that may bias the research done in this dissertation. And specifically focusing on the variables used along the several experiments. As a summary the following list shows all of the variables that have been used:

1. **Bug life cycle**:
   (a) Overall commit activity: this is the basic piece of information found in the SCMs
   (b) Bug fixing commits: subset of the commit activity
   (c) Bug seeding commits: subset of the commit activity
   (d) Developer: author of the source code change and associated to a commit
   (e) Time of commit: time when the commit was done
   (f) Files: those are affected by source code change in a commit
   (g) Lines of source code: lines touched by a developer in a commit and associated to a file
   (h) Programming language: this is associated to a file by its extension
   (i) Time to fix a bug in the SCM: time that passes between the bug seeding and bug fixing commit
   (j) Open report: piece of information found in the BTS, it is associated with the bug fixing commit by the number of the registered issue.
   (k) Close report: final step in the life of a bug in the BTS (although this may be opened)

(l) Time to fix a bug in the BTS: time that happens between the opening and closing report actions

2. Human Factors: **Experience**

   (a) Overall commit activity

   (b) Bug seeding commits

   (c) Bug fixing commits

   (d) Developer

   (e) Bug seeding ratio: this is defined as a percentage and calculated as the number of bug seeding commits out of the total activity in a given period.

   (f) BMI coefficient: this is defined as a percentage and calculated as the number of bug fixing commits divided by the number of bug seeding commits in a given period.

   (g) Territoriality: this is defined as the number of files only touched by one developer in a given period

   (h) Difference in type of activity: this is defined as the difference in are between the aggregated value of the fixing commits minus the aggregated activity of the seeding commits.

3. Human Factors: **Timeslot analysis**

   (a) Overall activity

   (b) Bug seeding commits

   (c) Bug fixing commits

   (d) Bug seeding ratio

   (e) Developers

   (f) Core and non-core developers: core developers are defined as the percentage of the community that usually undertake the 80% of the total amount of work.

   (g) Hour of commits: this is defined as the hour of the day where a commit is done

The following list describes the limitations to the study that have been found for each of the aforementioned variables. In some cases, some variables depend from some others. Thus, those are also affected by the limitations in their *parents*:

1. **Time to fix a bug in the SCM and in the BTS:** this is controlled by the *local time* provided by the author of the commit when submitting the change to the source code. As seen in 4.3.1, there is a minor set of the developers where the time

when a bug was fixed, was detected as happening earlier than the bug seeding action. This clearly shows a limitation when retrieving this information. The local time is stored in the SCM when the commit is done by a specific developer. However, it is not possible to control if that local time was real. This problem could also be enumerated for centralized SCMs such as CVS or SVN. If there is any problem or distortion in the server time. This leaves this issue as open question for the future, where a more controlled way of checking the local time of the developers should be taking into account, or at least, calculate the error of using this approach.

This problem also appears in the case of the bug tracking system, where some bugs have been detected to be closed before they were opened. Even in this case, where the BTS is a centralized way of retrieving information, this problem also happens.

In any of the cases, those *oddities* were removed from the dataset.

2. **Developer**: the characteristic use of developers done by the Mercurial repository makes this to be the *real author* and not the *committer*. As specified, by definition of core developer, those are in charge of the 80% of the total number of changes. However, this value may seem to be overestimated if compared to other communities since in other cases there is a differentiation between committer and author (as in the case of Git) or even there is not differentiation (as in CVS or SVN).

3. **Bug fixing commit**: these are the commits that have fixed an issue. As studied, there are few commits detected as being fixing an issue that are not really fixing an issue A.3. However, there could appear cases where a detected bug fixing commit is not such. The heuristic used, similar to previous work in the field [Kim *et al.*, 2008c], is detected as not being perfect, however the validation process has raised a high precision and recall of the proposed method.

4. **Bug seeding commit**: these are the commits related to a bug fixing commit. The *seeding* action has been declared as the involuntary moment when a bug has been introduced in the source code. Limitations to the method used have been declared as further work at 6.2 and filtered for the study in the specific experiments in 4.3.3 (chapter 4). In addition, since the detection of bug seeding commits are dependent on the detection of the bug fixing commits, the limitations found in the retrieval process of bug fixing commits are also applicable to the detection of bug seeding commits.

   In addition, this detection of bug seeding commits is done thanks to the existence of a *diff tool* that provides differences between each pair of revisions. However, as specified by [Canfora *et al.*, 2007], [Zimmermann *et al.*, 2006] and [Canfora *et al.*, 2009] there are some limitations with this type of studies.

5. **Bug report**: the selection of the *bug report based on the study of the log message* left by developers when committing changes to the source code have been declared as not being totally reliable [Antoniol *et al.*, 2008], [Bird *et al.*, 2009a], [Nguyen *et al.*, 2010], [Bachmann *et al.*, 2010]. However as specified in chapter 2, section 2.1.1, the bug prediction models elaborated still produce similar results when improving the method.

   In addition, since the creation of prediction models is out of the scope of this dissertation, this is not a key point to be resolved. Even more, the potential filtered of data due to this bias in the retrieval process, would at most select a subgroup of the initial dataset, what it is not a big issue.

6. **Territoriality**: the selection of this metric was initially proposed for the whole life of a project. However, the longer the life of a project, the smaller the set of files only touched by one developer. Thus, the definition of territoriality, although it is clear, evolves over time and should be taken carefully in older projects.

7. **Difference in type of activity**: the calculation of areas was made by calculating the resultant regression line of the total aggregation of lines. This is an approximation and is also a threat to validity when characterizing developers.

In addition, to the potential threats to validity found in the several variables used along this dissertation, [Fernandez-Ramil *et al.*, 2008] uses another set of construct validity aspects more general that should be mentioned and clarified in this section:

- **Incomplete or erroneous records**: although it is claimed by [Fernandez-Ramil *et al.*, 2008] that the use of the log messages left by developers is not reliable enough. It has been observed a pretty low level of unreliability with respect to the use of this piece of information in the case of the Mozilla Foundation. Thus, this is still a construct validity. However, as seen in the validation appendix (section A.3) the reliability when detecting fixing commits in the case of the Mozilla community is pretty high and provides confidence enough. In addition, the same heuristics used along this dissertation has been used in previous studies [Kim *et al.*, 2008c].

- **Biased samples**: as claimed by [Fernandez-Ramil *et al.*, 2008], the selection of random projects help to avoid the risk of having too many projects in specific phases of development. However, as seen in the life of the studied projects, there are projects in almost any stage of development. From pretty old projects such as the Mozilla Central project to others with pretty low history. However, it is true that the focus on the Mozilla community may affect the results since all of the projects are using a similar "bureaucracy". However, this can be seen from another perspective, the

better understanding of a specific community help to study specific processes carried out by developers when improving the development process, as has been detected in several results. (For instance, the analysis of time to fix a bug and its comparison with the definition of core and non-core projects).

- **Errors in data extraction**: as claimed by [Fernandez-Ramil *et al.*, 2008], the data extraction phase is complex and may be error prone. Some assumptions may have been done and they should be specified.

  With this respect, the data extraction process have been mainly detailed in chapter 3, where the main assumptions have been indicated. However, the main ones will be summarized again in the following list:

  1. **Use of *unified diff***: the BlameMe tool uses the unified diff to compare the differences between two versions of a file.
  2. **Detection of fixing commits**: those commits where the developer left a message following the heuristic: "B‖bug" + integer.
  3. **SZZ algorithm**: this dissertation is using the approach of the SZZ algorithm to detect the commit where the bug was involuntary introduced in the source code.
  4. **Detection of bug seeding commits**: those are based on the SZZ algorithm.

  In addition, the data extraction may contain errors not identified in the testing phase of the tool.

- **Data extraction conventions**: no data extraction conventions have been used in this phase. However, this is still something to be defined in the empirical software engineering field.

5.1.2 Internal Validity

The following internal validity aspects have been detected:

1. **Tools**: In general terms, the tools developed and used through this dissertation are probably far from being perfect. As software evolves, new requirements will arise and bugs will appear during their life cycle. However this is a potential problem when working in a fully empirical environment. Regarding to the tools, the one specifically developed in this dissertation has been tested from different perspectives that assure the quality of the data stored in the databases. With respect to the rest of the tools, those have been extensively used in other environments what improves the reliability of the software.

2. **Large set of scripts:** the use of large set of scripts to obtain the data may contain errors that are initially unavoidable. In any case, for each new script created, several manual tests were done in order to check the expected behavior.

3. **Granularity of the study**: as claimed by [Fernandez-Ramil *et al.*, 2008], the evolutionary behavior of the total system may be different from the subsystems and even more at the granularity of developers. In order to deal with this internal validity, several studies have been carried out at different granularities. However, it is clear that working at any granularity is not scalable. Thus, this dissertation has tried to work at different granularities depending on the type of study to better understand the small details that may offer finer granularities.

4. **Initial development:** this has previously been detailed as a construct validity, but this could be seen as an internal validity of the dissertation. The not randomly selection of projects may bias the final set of results. However, this has been avoided with the selection of small projects to bigger ones in the Mozilla community.

5. **Confounding variables and co-evolution**: as claimed by [Fernandez-Ramil *et al.*, 2008], specific variables may have not been studied and may affect the current set of results. For instance, in chapter 4, section 4.6, the selection of three ways of measuring experience based on previous research, does not avoid that there may appear other ways of measuring experience. Even more, all of that metrics can be affected by third part metrics that are not studied in this dissertation. Along this dissertation, the study of all of the projects together, but not aggregated has helped to avoid this type of effect.

5.1.3 External Validity

Several external validities have been detected when aiming to extend this analysis to other FLOSS or proprietary communities:

1. **Use of Author instead of Committer**: by default the Mercurial repository provides the author for each of the changes done in the source code. Compared to other repositories, this could produce different results. For example, centralized repositories such as SVN or CVS only provide the information of the person who uploaded the changes to the source code. The author is recognize in this case in the message left in the log or in a specific file (usually named as AUTHOR(S)). In other cases such as the Git SCM this information has to be added by the final committer in specific fields. In the case of the Mercurial SCM, this information is automatically retrieved in the source code change.

2. **Local time of commit**: the final set of studies related to the time of the day and the quality of the source code changes can not be extended to other communities, except if they are using a distributed SCM. The use of centralized SCMs (unless this information was also added) makes impossible to know the local time when the developer submitted the change to the source code.

3. **Use of unified diff**: this is the most extended way of providing diff information when looking for the difference between two files. However, other ways of providing diff information are possible, but not supported by the BlameMe tool. In addition, this tool nowadays works with Mercurial and partially with Git. The extension to other types of SCM implies the development of their specific parser.

4. **Relationship between fixing commit**, its log message and the bug tracking system: one of the main assumptions done in this dissertation is the easily identification of bug fixing commits when analyzing projects from the Mozilla community. This is thanks to the strict way of identifying information in the log message when committing a change to the source code. This type of simple heuristic must be change in order to reach a bigger range of actual fixing commits. In addition, a later validation of the heuristic should be done similarly as done in this dissertation.

5. **Programming languages**: the selection of programming languages used in this dissertation are initially filtered in the BlameMe tool by means of a regular expression. This could be modified in order to add or subtract programming languages, but this can also be done in the database when obtaining the data. However it is highly recommendable to be done in the first stages of the retrieval process modifying the BlameMe tool.

6. **Number of projects analyzed**: the use of the Mozilla community as the case study implies a better understanding of this community. However, this implies that the idiosyncrasy of the community is directly observed in the final results (for instance when the definition of core project affects the time to fix a bug decreasing it). A higher number of projects would enrich the study.

5.2 Discussion

5.2.1 Methodological Discussion

In general, one of the main points of discussion is focused on the method used along this dissertation. As seen in the analysis chapter, although the methodology proposed have been used in other papers and accepted in well known journals, this tends to underestimate results.

As a reminder, the main assumption of the methodology is based on the selection of the lines that take part of a fixing change. Selecting those that were modified or removed are the ones that were part of the buggy change. Thus, if someone needs to check the point in time where the bug was introduced in the source code, those lines are trace back in their history till their previous change.

The previous change (named along this dissertation as "buggy" change, bug injection or bug seeding commit) or previous changes (since it is possible that part of the lines were modified in more than one previous commit) is expected to follow a chronological time where in first place the bug seeding commit happens and later the bug fixing commit is done. In the meanwhile between those two events (or more than two in the case of more than one previous bug seeding change), some people from the community such as developers or users would have realized that there is a problem in the source code opening a bug report. Thus, the expected bug life cycle is as follows: a buggy change is done in the source code, a bug report is open in the BTS and finally that issue is fixed in the SCM and closed its correspondent bug report in the BTS.

However, along this dissertation it has been proved that in some cases the expected time of involuntary seeding a commit in the source code occurs later than the time of opening a bug report. Thus, this opens several methodological questions that should be answered before continuing with the rest of the discussion chapter.

Those questions can be, in general terms, summarize in the following three discussion questions (**DQ**):

1. **DQ1**: Is necessary to filter the dataset in order to select only the bug seeding commits that fit with expected chronological order of the bug life cycle?
2. **DQ2**: Are the previous analysis and results automatically invalidated?
3. **DQ3**: Is there anyway to improve the method in order to obtain more accurate results?

Regarding to the DQ1, the answer is negative. This basically depends on the type of study carried out. By definition of bug seeding commit (and this is the event of involuntary introducing a bug in the source code), a developer is introducing an issue in the source code that later must be fixed. As seen, this dissertation is divided in three main sets of experiments:

1. **Bug life cycle**: study of the time to fix a bug in the SCM and BTS, distributions of the datasets, demographical study of the life of the bugs, estimation of finding rates, defective fixing changes and number of previous bug seeding commits.
2. **Human related factors**:

(a) **Experience and bug seeding ratio**: introduction to bug seeding ratio, analysis of the evolution of the bug seeding ratio, territoriality, experience and characterization of developers.

(b) **Behavioral Patterns of activity in a 24 hours framework**: distribution of the changes, time of the day more prone to be buggy, experience and 24 hours framework of activity and comfort time of work.

In the first set of experiments, the study of the bug life cycle, several experiments were carried out. In all of them, the very existence of the bug seeding commit was the central point of study, except in the case of studying the time to fix a bug where it was necessary to filter the dataset in order to avoid those bug seeding commits happening later than the even of opening a bug report in the BTS. This improved the study only focusing on those bug seeding commits that happened before in order to calculate the time to discover a bug and the time to fix a bug. With respect to the rest of them: distributions of the datasets, demographical study and others, the time when a buggy change was done is not important at all, but the existence of the buggy change itself. Thus, for the rest of the experiments in this first set, the chronological order of the bug seeding actions is not necessary to be taken into account.

For the second set of experiments, the existence of bug seeding commits happening later than the correspondent opening report, does not affect the final results. For instance, the bug seeding ratio, base of the second set of experiments only measure the number of bug seeding commits. It is clear that specific previous bug seeding commits are not registered with this method, but the ones that are detected are still buggy changes. Thus, does affect the chronological order of the bug seeding commits for this set of experiments? the answer is again negative: it does not affect the final results. In any case, the values may have been underestimated.

Finally, for the third set of experiments, the experiments are again focused on the existence of the bug seeding commit and the moment of the day when this happened. Thus, the chronological order of the buggy actions do not affect the final results. Again, the results may be underestimated, but this is further work that would improve the proposed method.

Regarding to the second discussion question, the results are not invalidated. As detailed in previous paragraphs, in some cases the results may underestimate results. In the case where that assumption is an actual requirement, the commits that do not fit with the expected behaviour were ignored.

Finally, with respect to the third one, it is clear that there is room for improvement. In first place, the method and tools are probably far from being perfect and the diff tool is not prepared to calculate all of the information needed along this dissertation. However,

other proposals such as the "ldiff" to obtain more accurate results have been studied and due to time requirements not implemented as will be seen in further work section.

5.2.2 Results Discussion

The results in general have proved how the study of the event of bug seeding commit is important from several perspectives. In the case of this dissertation, the goal was initially study how this event takes place in the bug life cycle to be later studied from the software maintenance point of view.

Specific results such as the total time to fix a bug compared to the data obtained in the bug tracking system, will help in the effort estimation studies. As said, the total time since this is discovered till this is fixed has been estimated in around a 10%-40% of the total time, while the rest of it has been studied to take from a 60% to a 90% of the total time. Thus, this introduces another step in the study of the bug life cycle. So far, several studies focused on the estimation of time to fix a bug, on which bug will be fixed and in reliability software studies. However, this addition improves these well known fields of study.

As an example of the potential applicability of these results, the following can be cited, but others may be added. This list should be seen as an introduction, since the software engineering field is massively large. For instance, no mentioned is done to the software reliability field, although partially some of the results would help to tune some of the metrics (such as the time to fix a bug).

- Improving initial steps of any quality model: as specified by several quality models, there exist specific stages on the development process that includes the risk analysis. Among others, as seen in [Kan, 2002], the "Spiral Model of the Software Process" or the "Iterative Development Process Model" where this stage is specifically detailed. It will not be the same to work on an project where the issues last to appear a 90% of the total life of the bug, than those where take much less to be detected. Thus, this is a risk that projects may face and should be taken into account. In the case the times to detect a bug and fix it tend to increase, special actions should be done in order to improve the software maintenance phase.

- Helping in the "The Defect Prevention Process": as specified by [Kan, 2002], this is not exactly a quality model, but aims to improve the software development process. As seen in chapter 2, this process is focused on three main steps:

  1. Analyze defects or errors to trace the root causes
  2. Suggest preventive actions to eliminate the defect root causes
  3. Implement the preventive actions

As expected, the set of results of this dissertation specifically may help in the first bullet point: "...trace the root causes". Among others, specific human related factors have been studied.

- Improving quality process models: the use of maturity process models among companies is largely used in the industry. Partially, the results of this dissertation may help to reach specific levels of process maturity. Specifically, to those related to the quantitative analysis and obtention of metrics. Among others, it can be cited the SEI Process Capability Maturity Model and the later developed CMMI model that integrates practices from four CMMs: software engineering, system engineering, integrated product and process development and for acquisition.

  In this model, there are five levels of maturity: Initial, Managed, Defined, Quantitatively Managed and Optimizing. Specifically, the results of this dissertation may help in the risk management process areas (maturity level 3) and in the level 4 of maturity where the process areas are the organizational process performance (by means of measuring the general bug introduction rate of the organization among other possibilities) and the quantitative project management (where specific quantitative metrics to deal with the roots of issues could be added). In addition, some help in the level 5 (optimizing) could be provided, specifically in the causal analysis and resolution.

  Finally, since the study of the potential applicability of the results raised in this dissertation is out of the main scope of it, this should be taken into account as further work.

- Improving current effort estimation models: the study of the time to fix a bug and the typical function followed by the rate of finding bugs, may help when assessing potential effort in the following months in a project. Thus, from a management point of view, the study of this type of results will help when estimating cost and people.

Regarding to the results themselves, specific results have been compared to previous literature obtaining different results in some cases.

In first place, the distribution to fix a bug in the SCM or BTS has been tested to be significantly different. However, in any of the cases, divided per project and per data source, the distributions are not following a normal or log-normal distribution. This result is also a confirmation of the result obtained by [Gokhale & Mullen, 2010], where the authors claimed that the normal and log-normal distributions are not the correct one. However, since the study of the distributions are out of the scope of this dissertation, no further analysis has been made in this field.

In second place, the study of the experience from a quantitative point of view has brought three ways of measuring experience: general activity, fixing activity and ownership. The relationship between experience and quality of the changes has been previously studied as seen in chapter 2, section 3.4.4. Initial results showed that in some cases, there is a direct relationship between experience and quality of the changes(e.g.: as seen in [Eyolfson *et al.*, 2011]. However, results obtained in chapter 4, section 4.6 indicate that there is not a direct relationship between experience and at least the relative number of "buggy" changes done in the source code.

Finally, in third place, the study of the effect of the time of the day when introducing errors in the source code has been briefly studied in the literature. Only few authors has dealt with this idea and the results that can be compared reach similar conclusions. As seen in [Eyolfson *et al.*, 2011], the time of the day affects the quality of the changes. The evening and late night hours have raised as being potentially more "buggy" than the traditional working hours (*office time*). Even when in both cases the case of study was FLOSS projects where there is always a hybrid community of people paid and volunteers.

5.2.3 Other Discussion Points

In addition to the general discussion

1. **Question: What about the existence of other bug-seeding commits? or at least other areas of a file that could be affected, in a middle commit between the bug seeding commit and bug fixing commit?** This dissertation is using the basic assumption, as being accepted in the literature that the lines that were removed or modified in a given commit (detected as fixing an issue) are the ones guilty of that bug. Thus, tracing back the history of those lines, the answer to the question who, when and what is obtained. However, several limitations have been seen during the whole process of this thesis. Although the goal of this is not to improve the process of detection of bug seeding and bug fixing commits, but the better understanding of the bug fixing process and some reasons why a bug is introduced in the source code. This point will discuss the several limitations found with this respect:

   (a) **Is a bug seeding commit the actual one where the bug was introduced in the source code?:** The results seen in 4.3.1 have demonstrated that, partially, the method used in this dissertation can be improved in the future. Although the use of the method proposed in this methodology has been accepted as a way of calculating when a bug was introduced in the source code in the literature, it has been seen how there are bug reports that were opened before the detection of the bug seeding commit. Thus, bringing in context a strict

definition of bug seeding commit: "Time when a bug was introduced in the source code", the results have shown that a subset of the bug reports were opened before the bug was even introduced in the source code!. Clearly, this is a too strict definition of bug seeding commit and this should be more flexible and there is still gap for improvement here.

(b) **Is the local time accurate enough for the results?:** In a pretty low percentage (lower than 3%) the time when a bug was fix happened before that when that bug was seeded. Since the time when a commit is done is stored based on the several developers time zone, this raises a limitation with this respect.

(c) **Is the one to one consideration when using bug-seeding and bug-fixing commits a good approach?** (discussion done in 4.3.1. This is a key assumption of the thesis. Most of the results are also based on this assumption. Other options were those taking only the first of those bug seeding commits, taking the very last one or using an average. In the first case and the latter case the values are closer between them than if compared to the other two cases. Indeed, it makes more sense to use at least an average or one to one than taking only one of the extremes. However, the final selected approach was to use the one to one approach. This perhaps is not a big issue in section 4.3, but it is when analyzing the pyramids of population 4.4. The main reason for this is based on the assumption that the bugs are seeded Finally, it is worth to remember that around a 40% of the total issues detected were seeded in more than one previous commit 4.5.

2. **Why was not "ldiff" used instead of BlameMe?**: in first place, *ldiff* is not a tool to retrieve information from each pair of revisions to create a relational database. And in second place, this only works in the specific case of renaming files and specific cases where lines are not correctly tracked. In addition, for performance issues, the ldiff command must be run between each complete full files (for each pair of revisions), what would increase the complexity in time as much as becoming hard to extend the study to other SCMs or even bigger projects. Only partially storing the diff between each pair of revisions, it is storing a small percentage. In addition, the time, if each pair of files is compared would be much longer.

3. **Why was not retrieved renamed files by BlameMe?**: although Mercurial is a distributed SCM such as Git or Bazaar, the information about renaming files can not be easily obtained as it is, for instance in Git. This makes difficult the retrieval process of the renaming files unless a more depth study is done.

4. **What happens when a bug is fixed in more than one commit?**: all of the fixing commits are taken and studied from an independent point of view. Thus, it means that in this specific case, those will be taken as different fixing commits and separately studied in all of the experiments done.

5. **Differences between previous bug fixing changes and current analysis**: with this respect, the authors [Kim *et al.*, 2006] stated that around a 30% of the changes were "buggy" changes out of the total activity. However, a basic analysis of the dataset, shows that in most of the analyzed projects, just using the bug-fixing activity in commits, those percentages raise a 60% of the total activity in the case of the Mozilla Central project.

5.2.4 Observations to the Mozilla Community

As seen, the Mozilla Foundation has developed specific quality assurance (QA) policies to work on key projects. The most important projects have been defined as "core" and an extra peer review process is carried out by more experienced developers.

This definition of core has made possible an improvement of the time to fix a bug in general, decreasing by 10X the amount of time if compared to the rest of the community projects (chapter 4, section 4.3). Thus, the existence of a specific policy in this sense has helped in the development phase, but also in the maintenance effort later undertaken.

However, some details detected along this dissertation make possible to study the peculiarities of its developers and the general maintenance effort. In first place, the defective fixing rate (chapter 4, section 4.5) has proved to be pretty high in "core" projects if compared to the rest of the cases of study. Thus, why is this happening?. A potential explanation is the continuous development process in some of them (specifically in the Mobile Browser project) that increases the general activity, but for some reason, also the relative fixing activity and defective fixing changes. Further work should be done with this respect.

In addition to the maintenance effort, it has been detected an improvement during the general life of the core projects. As seen in chapter 4, section 4.6.1, figure 4.19, the BMI coefficient is increasing in general in core projects, except in the case of the Tamarin Redux project. This indicates that the fixing activity is being higher than the seeding activity. In the case of the bug seeding ratio activity, figure 4.20 shows that results. However, this figure indicates that the relative number of buggy changes does not decrease in all of the core projects. For instance, it is clear than in the Comm Central project there is a decrease, but this is not happening in the Mozilla Central project. Even more, the Mobile Browser project slightly increases. Thus, although the number of fixing commits is increasing if compared to the bug seeding commits, it is not clear that the general tendency

when introducing errors in the source code is decreasing. In fact, the BMI coefficient is found under 100% in almost all of the project, what means that although this coefficient is increasing, the general ratio of buggy changes is relatively being maintained stable. Thus, specific tasks should be done by the Mozilla community in order to increase the BMI coefficient and decrease the bug seeding ratio. Basically, this is directly related to the defective fixing changes (aforementioned) and the core projects show the highest ratio of defective fixing changes. As seen, this is probably balanced by the effect of fixing quickly the bugs found in the core projects.

Regarding to the timeslot analysis, extra care should be taken by developers during their non-typical working hours. It has been detected higher rates of bug seeding ratio during those hours where developers do not usually work. (chapter 4, section 4.7). Those changes may be affected by external conditions such as deadlines that may force developers to work under stressful conditions.

5.3 Results

This section summarizes main results found in chapter 4. Results are divided into three main parts:

1. **Bug life cycle**: section 5.3.1 will detail main results regarding to the bug life cycle and time to fix a bug.
2. **Human factors - experience**: section 5.3.2 will detail main results regarding to the human related factor of experience.
3. **Human factors - timeslot**: section 5.3.3 will detail main results regarding to the human related factor of time of commit.

For each of the sections, the hypotheses tested will be summarized adding the correspondent "rejection" or "fail to reject" of the null hypothesis. In addition, a list of lessons learned will be added together with other experiments whose result was not based on a null hypothesis.

5.3.1 Bug Life Cycle

This section aims to show main results in the bug life cycle when introducing the concept of bug seeding commit. This new event has provided a general glimpse from the very beginning of an issue, where this has been "seeded" till the end of its life where this is fixed.

As a summary, the four main variables or attributes used along this set of results are the bug seeding and bug fixing commit and the open and close report.

ID	Section	Test applied	Result
$H_{0,0}$	4.3.5	Time to fix a bug obtained from the source code and from the bug tracking system follow the same distribution	Rejected
$H_{0,1}$	4.3.5	Time to fix a bug obtained from the source code follows a normal distribution	Rejected
$H_{0,2}$	4.3.5	Time to fix a bug obtained from the source code follows a log-normal distribution	Rejected
$H_{0,3}$	4.3.5	Time to fix a bug obtained from the BTS follows a normal distribution	Rejected
$H_{0,4}$	4.3.5	Time to fix a bug obtained from the BTS follows a log-normal distribution	Rejected (except in Venkman project)

Table 5.1: *Null Hypotheses Results: Time-to-fix-a-bug Distributions*

This section is as follows: subsection 5.3.1 will detail the null hypotheses tested and their results, while the rest of the sections will depict the major lessons learned from each of the analysis done.

Null Hypotheses Tested: Bug Life Cycle

The null hypotheses defined at chapter 4, section 4.3, are detailed with results in table 5.1. In all of the cases the null hypothesis is rejected. Thus, the distributions of time to fix a bug in the case of the source code and in the case of the bug tracking system and neither following a normal nor log-normal distribution.

Only in few cases, such as the Venkman project, the time to fix a bug in the BTS is following a log-normal distribution.

The study of the populations indicates that the distributions are not following a normal nor log-normal distribution. In addition, the distributions among different data sources have been studied. The results indicate that also in this case the distributions are different.

Lesson learned: *the distributions of time-to-fix-a-bug in the SCM and in the BTS are following different distributions. In any of the cases, the dataset is following neither a normal nor log-normal distribution.*

Time to Fix a Bug: Lessons learned

The time to fix a bug has been defined as the difference between the time when a bug has been introduced in the source code and the time when that bug has been fixed in the source code. In addition the time to fix a bug in the BTS has been analyzed.

The results show (chapter 4, section 4.3.1, table 4.14 and table 4.15) how there are projects where the mean and median are close to dozens of days, while there are others

whose time to fix a bug tend to be around hundreds of days (even reaching the thousand days). These results do not respond to the size of the project or number of developers, but to the policy developed by the Mozilla community. This policy, as a reminder, specify some projects as being "core". This definition of core implies that more resources from the community are spent in order to double check the changes to the source code.

Thus, those projects with a declared policy of being "important" (core) for the Mozilla community are the ones with a lower mean and median time to fix a bug in the source code. This points the importance of defining concrete policies in the communities (at least the largest ones) where the movement of resources (in this case a peer review policy in the changes in the "core" projects) is a key point of the goals of the community.

These results are also confirmed by the values obtained when studying the issue tracking system based on the bugs obtained from the Mercurial SCM (chapter 4, section 4.3.1, table 4.3.2). In this case the mean and median is in the range of days for the core projects, while for the rest of them this range increases to the dozens of days.

Lesson learned: *the definition of policies from the community help in the fixing process and reduces the time to fix a bug. In general terms: the definition of quality process policies helps in the software maintenance process.*

In addition, the study of the whole life cycle of the bugs have raised new datasets regarding to the total time that they remain in the source code. In first place since they are *seeded* until they are discovered. And in second place since they are discovered until they are fixed. It has been estimated the total time to discover a bug as a 60% to a 90% of the total life of a bug.

Lesson learned: *the time that takes to discover a bug since this has been seeded in the source code, has been estimated in a 60% or 90% of the total life of a bug. Projects with core policies tend to show value closer to the 60% of the total time.*

Finding Rate of Bugs: Lessons learned

The very existence of bugs in the source help, when they are discovered, to study how fast they are being found by the members of the community.

The results have shown that in most of the cases, the rate of finding bugs follows a lineal or exponential model. In terms of finding rate, the exponential model means that most of the bugs are found at the very beginning of their life, so, there are less pretty old bugs if compared to the lineal model.

In any case, the existence of these models, where the fit values are over 0.8, as seen in figures 4.14 and 4.15 (chapter 4, section 4.4.2) help to understand how developers discover bugs.

In this case, the definition of core or non-core project does not seem to alter this finding rate, except for the main project of the Mozilla Foundation: Firefox, which is the only

one where most of the studied data have followed an exponential model.

The potential applicability of these results are clear: this could help in the estimation process for costs in the software maintenance phase.

Lesson learned: *the finding rate shows that most of the finding bugs process follow a lineal or exponential model. Most of the projects and set of bugs are found following a lineal model.*

Defective fixing commits: Lessons learned

The defective fixing commits are defined as those fixing commits that later have introduced new errors in the source code.

The percentages go from a 27% in the case of the Camino project to a 87% in the case of the Thunderbird project. In this case, the policy of core project is the opposite of these percentages. Those projects defined as core are the ones with a higher percentage of buggy commits.

Lesson learned: *the most important a project is for the community, the higher the defective fixing commits*

This lesson learned may change the perception that the higher the number of defective fixes, the worse the bug fixing process carried out by the community. From another point of view this may mean that the higher the activity carried out by a community, the faster its evolution and, thus, the higher the number of changes that are later modified.

In addition, it is worth mentioning the fact that those projects that initially seems to be driven by people from companies, such as those dealing with ActionScript (actionmonkey or tamarin family projects) show a lower ratio of defective changes in the source code. This may point to the idea that community driven projects are more prone to be "buggy", however this, again, may point to the faster iteration in those type of projects.

5.3.2 Human Factors: Bug Seeding Activity and Experience

Once the bug seeding event has been studied from different perspectives, and having specific information from each of the events when a bug has been involuntary introduced in the source code, the causes why they were introduced were studied.

As identified in the first chapters of the dissertation, there could be specific causes why an issue is introduced in the source code. This thesis is focused on the potential activity done by the community and potential human factors that may imply introducing errors.

Among others, the tiredness, due to deadlines, experience, territoriality or other similar factors are studied. Those are all related to the information that can be found in the SCM, such as the time when a commit was done, timezone, author, files touched, lines added, modified or removed and others.

ID	Section	Null hypothesis	Result
$H_{0,0}$	4.6.1	The bug seeding ratio of the developers follow a normal distribution	Rejected
$H_{0,1}$	4.6.1	The older the project, the higher the bug seeding ratio	Rejected
$H_{0,0}$	4.6.2	There is not correlation between bug seeding ratio and number of commits	Fail to reject
$H_{0,1}$	4.6.2	There is not correlation between bug seeding ratio and territoriality	Rejected
$H_{0,2}$	4.6.2	There is not correlation between bug seeding ratio and fixing activity	Rejection depending on the project
$H_{0,3}$	4.6.2	There is not correlation between number of commits and territoriality	Rejected
$H_{0,4}$	4.6.2	There is not correlation between number of commits and fixing activity	Rejected
$H_{0,5}$	4.6.2	There is not correlation between number of territoriality and fixing activity	Fail to reject

Table 5.2: *Null Hypotheses defined in section 4.6 and their final results*

This section will present an initial list of potential causes that may provoke "buggy" changes and correlated them with the bug-seeding ratio, which is the number of bug seeding commits out of the total activity undertaken by a developer.

Subsection 5.3.2 shows a summary of the null hypothesis tested for the population of study. The rest of the subsections detail the lessons learned from the Analysis chapter.

Null Hypotheses Tested: Human Factors - Experience

The null hypotheses tested along this set of results are summarized in table 5.2.

Bug Seeding Activity and Experience: Lessons learned

This subsection aims to present the several hypothesis testing carried out and their results.

The bug seeding ratio has been introduced in this section of the analysis chapter and defined as the number of commits detected as being buggy out of the total number of commits.

The bug seeding ratio has been defined as the number of commits detected as being buggy out of the total number of commits. This value has been depicted in figure 4.17 (chapter 4, section 4.6) where the higher number of developers tend to be focused along a vertical line following a gaussian bell.

From an overall perspective for each of the projects, and studying the evolution of this bug seeding ratio, figure 4.18 depicts the results.

As expected by the Lehman's law, the source code tend to decay, and this makes the source code less maintainable. However, it has been seen, how the bug seeding ratio does not increase. In fact, as seen in the previous aforementioned figure, only projects such as the Firefox project (Mozilla Central) presents a slightly increase in the bug seeding ratio. However, in other projects, such as mobile-browser or Tamarin Central the bug seeding ratio remains stable and even more, in some others such as the ActionMonkey or Tamarin Tracing tend to decrease.

Lesson learned: *the age does not seem to be a definitive indicator of software maintenance. Neither the size of project nor the developer team.*

Characterization of Developers: Lessons learned

The calculation of bug seeding and bug fixing ratio per developer and aggregated by time, shows a specific pattern of activity that makes possible a division of them based on the difference between all of them.

Figure 4.28 (chapter 4, section 4.6.3) shows this data where the median is close to 0. This means that explaining this dataset by the median, the differences between the bug seeding and bug fixing ratio is close to 0. Thus, the activity, in terms of fixing activity is balanced with the involuntary introduction of bugs.

If those areas are deepened studied, there are two main sets of developers: those whose main activity is focused in maintenance effort and those whose main activity is other than maintenance (generally adding functionality, but not fixing commits).

The differences show that the bigger areas are those found at the bottom of the figure at the top. Reaching differences of 150,000 units, while the others reach differences of only 50,000 units (pure maintenance activity). Thus, the differences in area is smaller (up to three times) between the pure maintainers and the rest of them.

Lesson learned: *the characterization of developers is possible using the differences in areas between the bug fixing and the bug seeding activity. This provides an approach to identify the "pure" maintainers of the community.*

Experience and Bug Seeding Ratio: Lessons learned

As seen in chapter 2, section 3.4.4, the experience or expertise of developers have been quantitatively measured in three different ways:

1. Number of commits: this has been measured as the number of total commits done.

2. Number of fixing commits: this has been measured as the number of bug fixing commits.

3. Ownership of the source code: this has been measured using the definition of territoriality.

This three ways of calculating experience, has been correlated to the bug seeding ratio, defined as the number of bug seeding commits out of the total activity.

The results indicate that in general terms there are not direct correlations among the different ways of measuring experience and the bug seeding ratio. However, there have been found a correlation between the bug fixing activity and the total activity.

Lesson learned: *There is not empirical evidence that the experience of the developer, in any of the studied ways, is correlated to the bug seeding ratio.*

As a reminder, experience has been measured counting commits, fixing commits or measuring ownership in the source code. Thus, the following list detailed the several analysis done and their results:

- *There is not empirical evidence that there is a correlation between the bug seeding ratio and territoriality.*
- *There is not empirical evidence that there is a correlation between the bug seeding ratio and number of commits.*
- *There is not empirical evidence that there is a correlation between bug seeding ratio and number of fixing commits.*

In addition, the potential relationship among the types of measuring activity have been analyzed.

Lesson learned: *There is not empirical evidence that there exist a relationship between territoriality and the number of commits.*

Lesson learned: *There is not empirical evidence that there exist a relationship between territoriality and fixing commits.*

Lesson learned: *There exist a direct relationship between the number of commits and the bug fixing commits.*

5.3.3 Human Factors: Timeslot Studies

Finally, the potential effect of the working hours was studied from different perspectives.

Null Hypotheses Tested: Human Factors - Timeslot

The null hypotheses tested in this set of experiments can be found in table 5.3.

ID	Section	Null hypothesis	Result
$H_{0,0}$	4.7.1	The annual activity of each project is similar	Rejected
$H_{0,1}$	4.7.1	The overall activities in the Mozilla are normally distributed	Rejected
$H_{0,2}$	4.7.1	The overall activities in the Mozilla projects are similar	Rejected
$H_{0,3}$	4.7.1	The bug-seeding activities in the Mozilla projects are similar	Rejected
$H_{0,4}$	4.7.1	The bug-seeding and overall activities in the Mozilla projects are similar	Rejected
$H_{0,0}$	4.7.2	The bug seeding ratios are similar among projects	Partially rejected
$H_{0,1}$	4.7.2	The bug seeding ratio is affected by the time of the day	Fail to reject
$H_{0,2}$	4.7.2	The bug seeding ratio is uniform along the day	Rejected
$H_{0,0}$	4.7.3	There are not differences in the bug seeding ratio between core and non-core developers	Rejected
$H_{0,1}$	4.7.3	The non-core developers bug seeding ratio is greater than the core developers'	Fail to reject

Table 5.3: *Null Hypotheses: Timeslot Studies*

Distribution of the Effort: Lessons learned

Distributed teams of developers working along the world, also add another variable to the study, they usually work in different times of the day. Some of them may work during a more traditional time of the day, while others work after dinner or even during late-night hours.

Thus, the distribution of effort is, in first place, distributed along the world and, in second place, distributed along the 24 hours of the day.

Figure 4.29 (chapter 4, section 4.7.1) shows these results where there are always two or three humps of activity. However, the study of the distributions shows that in all of the cases are different.

Lesson learned: *The distribution of effort is divided in two main sets. The first one during the morning/afternoon timeframe, while the second one, from the afternoon onwards.*

Lesson learned: *The distribution of effort among projects is not following the same distribution. Thus, the datasets can not be aggregated to be studied.*

Lesson learned: *The distribution of bug-seeding ratio shows a difference between those projects marked as "core" by the community and the rest of them. The higher the difference between the maximum and minimum bug-seeding ratio, the younger the project is.*

Time of the Day: Lessons learned

Thus, the distribution of effort along the 24 hours of the day may provoke that specific commits could face extra risks when changing the source code. For instance, a developer working during late-night hours is expected to introduced, in average, more errors in the source code than a developer working during the light hours.

This expectation is confirmed by the results, where the highest peaks of bug seeding activity is found during the smaller humps of activity. The later is, the more possibilities to introduce an error.

Lesson learned: *The highest peak of bug seeding ratio is found during the timeframe after the "office" time.*

Core and non-core Developers: Lessons learned

Another variable to take into account is the experience. As seen in previous sections, the experience does not seem to be a key factor to introduce a less ratio of errors in the source code. This idea is corroborated by this study where there are not main differences between the core and non-core developers.

Lesson learned: *there is not empirical evidence of differences between the bug seeding ratio of core and non-core developers.*

Comfort Time: Lessons learned

Finally, this experiment has shown how activities out of the comfort time of the developers tend to be more "buggy" than the common pattern of activity. This has been depicted in figures 4.35, 4.36, 4.34, 4.33) (chapter 4, section 4.7.4).

Lesson learned: *Out of the common timeframe of activity, for a developer, there is a higher tendency to introduce more bugs in the source code.*

Chapter 6

Conclusions and Further Work

‘‘Let them (the foreigners) invent!’’
‘‘¡Qué inventen ellos!’’
Miguel de Unamuno

This chapter will detail the main conclusions based on the initial research goals and contributions claimed in chapter 1.

With this purpose, section 6.1 will conclude the achievements of the dissertation and further work will be detailed in section 6.2.

6.1 Conclusions

The consideration of the organizational structure and human factors as a potential source of bugs have raised interesting results that should be taken into account when managing distributed developer teams.

The use of the Mozilla community, with specific policies regarding to the maintenance process (peer review of the changes to the source code made in the *core* projects), has helped to provide specific data about how the software development process takes place.

Regarding to the initial set of research goals and contributions of this thesis introduced in chapter 1, section 1.2 and section 1.3, those were divided into two main sets: the study of the bug life cycle from several perspectives and the study of potential human related factors that may be more prone to introduce errors.

With respect to the first of them, the study of the bug life cycle, this dissertation has studied the *roots* of the errors, named as *bug seeding commits*. Using this as the first event in the bug life cycle, it was discovered how from a 60% to a 90% of the total time of a bug in the source code takes place since this is introduced until this is discovered. The rest of the life of the bug takes place in the bug tracking system where developers aim to fix it.

In addition, estimations to find a bug were studied in all of the projects in the Mozilla community. It was found how old bugs tend to remain for a long time in most of the projects. Besides, the finding of bugs usually follows a lineal distribution what could be useful when estimating the number of remaining bugs for a specific period.

Moreover, the study of fixing commits that introduce a new issue in the source code opens new research questions related to the big differences found: from a 27% to a 93%.

Finally, the characterization of developers based on the typical activity (*seeding* or *fixing* errors) in the source code helped to discover that there are pretty different behaviors when committing. In first place, the median of the developers seed as many bugs as they usually fix, what balances the total activity. In addition, it was found that the extremes are quite different: those that are detected as *pure* maintainers and those whose activity is mostly detected as being *buggy* with pretty or null activity when fixing errors. The differences found between the extremes are three times in favor of the developers that usually do not fix bugs. Or in other words, the latter set of developers hardly fix an issue, while the first set of developers (pure maintainer) usually introduce errors in the source code.

Regarding to the human factors that may be more prone to be buggy, it was claimed that may exist potential relationships between the experience of the developers and their bug introduction ratio. However, no actual relationship has been found. On the contrary, another similar study did find specific relationship what indicates that further discussion should be undertaken.

With respect to the time of the day activity, it was discovered how after a typical *office-time* working hours, there is a higher probability of introducing errors. In addition, isolated developers (the two most important in terms of activity per studied project) were studied looking for extra information with this respect. It was found that out of their typical working hours (either during the day or during the night), all of the ten studied developers increased their probability of introducing an error.

In this context, differences in terms of probability of introducing errors in the source code were studied between the *core* and *non-core* developers. However, as similarly obtained in the experience set of experiments, no differences were found at all. Thus, those two set of developers did not show differences in the probability of introducing an error.

Summarizing: as a reminder the initial goal of this dissertation was the study of the bug life cycle of the bugs, focusing on the bug seeding activities carried out by developers. In addition, the study of human related factors that could be more prone to be buggy was intended. With this respect, we can claim that the general goal was accomplished since specific studies in the bug seeding event were undertaken. This helped us to advance in the current state of the art related to the bug life cycle. In addition, the study of two

potentially buggy human related factors were studied. This allowed us to compare those with existing previous results and advance again in the state of the art, adding valuable information in the case of the study of the typical activity carried out by developers in a 24 hours framework of activity.

6.2 Further Work

Finally, this section aims to detail the open questions remaining along this dissertation that keep room for improvement

6.2.1 Methodology Further Work

The study of the bug seeding commits was based on the analysis of the source code lines. However, some peculiarities of the methodology proposed were raised and further work is proposed to deepen into them.

In first place, huge movements of lines have been detected as taking part in specific commits (as seen in subsection 4.2.2, where more than 800,000 lines were *touched* in just one commit. However, as indicated, this effect is diluted by the minimum piece of information that is used along this dissertation: a commit. In any case, those commits with such a big changes should be studied and better understand why developers are doing this and their effect in the general methodology.

In addition to the previous movements of lines, there are also specific cases where when an author is fixing an error in the source code, she is also adding new lines to fix it. As detailed in chapter 3, the SZZ algorithm, used as an assumption along this dissertation only takes into account those lines that were *modified* or *removed* and later this traces the *roots* of those lines. However, it is not clear what is the percentage that those *added* lines represents in each of the fixing commits.

One of the improvements that can be undertaken is the use of the *ldiff* tool that improves the detection of *modified* lines. However, it was decided not to use this approach since this would be too complex in time. As an example, the following paragraphs will extrapolate the time of calculation of the *ldiff* tool with the current context of this dissertation.

As specified by the authors [Canfora *et al.*, 2009], the conditions of calculation were a laptop with a 2Ghz CPU, plus 1 GB of RAM and a quadratic complexity. With this hardware specification, the time to compare 31 lines took 2 seconds and the time to compare 171 lines took 54 seconds.

In the following, let us assume a lineal and not exponential increase of the time to obtain new data. This provides, with the lineal approach a slope of 8.45 in the equation of the line $y = m * x + b$. Now, let us take the biggest change in terms of lines in the

comm-central repository (as an example) where the number of lines that were "touched" in commit 817 were more than 800,000.

With this basic approach the estimated time to calculate the differences, with the hardware specification takes around 79 days of calculation for one commit. Of course, it is necessary to remember that the hardware specification where the data of this dissertation is calculated is more powerful: 2 processors and 4 GB of RAM. However, even if the time of calculation is divided by 4, it is still taking "days" when obtaining data only for one commit.

Thus, although this method improves a lot the precision of the results, this was decided to not being implemented in the dataset because of its complexity in time.

Coming back to the open research questions as further work, the definition of bug seeding commits should be expanded to other commits. As seen in section 4.3, in some cases the time to fix a bug in the bug tracking system is higher than the data retrieved from the source code. This opens new questions related to the methodology used in this dissertation (also used in the literature) to specifically check the point where the bug was initially seeded.

Finally, it is necessary to mention the assumption done in the last set of experiments related to the effect of the time of the day in the quality of the changes. It was assumed that all of the days are similar. Thus, bank holidays, weekends and other similar cases are ignored in the general methodology when aggregating the data in 24 hours. Extra analysis should be done to enrich this approach.

6.2.2 Results Further Work

Regarding to the main results found, there are also specific points that are likely to be studied in the future.

In first place, the initial filtering of the dataset based on the programming languages was focused on traditional ones. However, others such as XUL, XML or HTML were removed since they can be developed using specific GUI tools that facilitates this process. However, in the specific case of XUL, this is a key part of the Mozilla architecture and specific analysis should be done in order to compare it with more traditional programming languages.

In addition, the study of the size of the changes was also left as further work due to space requirements. However, it has been studied as a potential source of bugs and its relationship with the bug seeding ratio is necessary.

The study of the relationship between experience and quality of the changes have addressed that there is not an actual relationship. However, other studies have confirmed this hypothesis. With this respect, further work should be done in order to look for other definitions of experience from a quantitative point of view.

Although the dataset is based on a distributed development team such as the Mozilla community, other projects and communities should be added to enrich the results. Through this thesis, it has been seen the behavior of the Mozilla community with its specific peculiarities (e.g.: specific policies defined for *core* projects) what may smooth the final results since the maintenance activities are well defined.

Also, the estimation of finding bugs in a given period has proved its potential applicability. However, in some cases the number of dots in the study were not enough to demonstrate results (in around a third of the total values). Using other granularities and selecting other communities with larger history may improve this approach. Since the cost and effort estimation is still a field of study and even more in FLOSS projects where there are not specific cost models, this may help to, at least, understand how fast the community work when discovering errors in the source code. Once this is understood, specific models may be developed to characterize the bug fixing activity in FLOSS communities (and other proprietary systems using a SCM).

In addition to the previous variables used, other authors have dealt with the idea of the organizational structure of the communities as a better way of estimating future errors [Nagappan *et al.*, 2008]. Those should be included in future studies.

Finally, the evolution of the bug seeding ratio and its correlation with other metrics such as complexity metrics could be studied. This may help to make a estimation of the increase of the complexity of the source code once the bug seeding ratio tends to show an increasing evolution. (Figure 4.18).

Appendix A

Validation

This appendix aims to validate the way the tool *BlameMe* retrieves the data from the original data sources.

In general, the three following bullets will describe the general overview of this chapter.

1. **General Overview**: for a randomly set of commits, check if the information found in the diff tool provided by the different distributed SCMs, is correctly stored in the database.

2. **Validation line by line**: Once the data is correctly stored in the database, it is necessary to check if that data is correctly cataloged. In other words, if the lines that take place in each chunk of information are correctly assigned to the specific files with the correct state (added, modified or deleted).

3. **Source of errors**: a potential set of errors is expected to be found. The goal of this item consits of the source of that error in the general methodology.

A.1 General Overview

As previously explained, for a given set of randomly selected commits, it is necessary to check if the data found at each commit is correctly stored in the database. Thus, it is tested if the information found in the differences found in each pair of revisions is stored in the database "as it is".

For this experiment, 20 commits have been checked and for each of them all of the lines that were added, modified and removed were studied. Also the number of unique files and extra information is checked.

The case study selected was the Firefox (mozilla_central repository) and Thunderbird community (comm_central repository).

Method.

The methodology to validate the results for a given commits is the following:

- query: select * from blame where commit_id=38448;
- hg diff -r38448 -r38449
- Compare values provided at the database with the specific diff directly in the SCM.
- For each of the lines, compare its state (ADDED, REMOVED or MODIFIED) with the data at the SCM.
- Check that no-source-code files are not stored in the database.
- Summarizing, next fiels are measured:
  - Number of files involved
  - Number of added lines
  - Number of removed lines
  - Number of modified lines
  - Compare results between the database and the hg command exit

The set of randomly selected commits is the following:

Firefox commits: 38448, 43510, 42707, 14939, 30568, 35096, 43111, 15858, 46490, 29640, 30966.

Thunderbird commits: 681, 3496, 1664, 4123, 1361, 1734 ,625, 553, 3249, 3584.

Results.

The results show that the comparison is satisfactory since for all of the commits exactly the same information was found in the database when retrieving that information. The only not evaluable case was a commit were 100,000 lines were handled, what made a manual analysis impossible.

In order to avoid too huge commits, only source code such as C, Python or C++ were taken into account ignoring those "tag" based languages such as HTML or XUL. Table A.1 shows the results of this validation.

As a conclusion, we could claim that the data found in the diff tool provided by the several SCMs is found as being the same that in the database.

Commits - Firefox	Result	Commits - Thunderbird	Result
38448	OK	681	OK
43510	OK	3496	OK
42707	OK	1664	OK
14939	OK	4123	OK
30568	OK	1361	OK
35096	OK	1734	OK
43111	OK	625	OK
15858	OK	553	OK
46490	OK	3249	OK
29640	OK	3584	OK
30966	OK	-	OK

Table A.1: *Commits selected as case study for the validation*

A.2 Line by Line Validation

The line by line validation aims to study the possible errors detected when using this approach at the level of lines. Thus, more than the tool validation, what it is being measured in this section is if the diff tool is providing the correct information for our purposes.

In a more detailed way, the diff tool is used to obtain the differences between two files. This tool is also used by the patch tool when applying a patch to the source code. This is in fact a diff with the new set of lines modified, removed and added from the original.

There are not important errors found for this two tools in an UNIX environment, however, this thesis is using this information in a bit different way. Specifically, this information is being used to follow the life of a line from one revision to another.

A.2.1 Database to Real Data found at the Source Code

This validation aims to check if for a given set of lines selected at the database, those are found in the specified file in the same position and with the same content that the one specified in the database. With this validation we could determine if the diff command is providing the correct information in each case. Thus, the lines obtained with the stored data, should be found at the source code tree (for a given file and commit).

The analysis is based on the randomly selection of ten lines and followed its life.

Method.

The method is based on the following steps:

- Check the commits where the line is detected to be added, modified or removed.

Line id	Result of Comparison
358840	OK
441214	OK
1130682	OK
501998	OK
2533611	OK
821186	OK
1253452	OK
2606155	OK
178110	OK
492626	OK

Table A.2: *Lines selected as case study for the validation*

- "cat" the given file for the given repository to check the situation of the line (eg: hg cat -r0 mailnews/local/src/nsPop3Service.cpp > output.txt).
- If not possible to cat the file, then diff between two revisions to check if the file was not created before that commit (what makes fail the second row of the table).

The several lines were all selected in the Thunderbird project.

Results.

Ten lines have been randomly selected and studied their results found at the database. For all of them, there are not inconsistencies found between the database and what we found at the source code. Table A.2 shows the results for each of the selected lines

A.2.2 Real Data found at the Source Code to Database

This section aims to validate the opposite data. For a set of lines randomly selected in the source code, check their evolution and test if their changes are being correctly stored in the database. With this validation it is expected to determine if the diff command is providing the correct information in each case. Thus, the lines obtained in the source code and their changes should be stored in the database.

Results

In some occasions, source code lines are modified, but the diff tool detects them as added lines. This is the case of a piece of source code where new comments have been added and at the end, some lines have been also modified. Those lines were detected as new lines, while some part of the comments have been detected as modifications of the old source code lines.

An example of this could be found in the following:

```
Initial file:
```

```
line 1
line 2
line 3
line 4
```

Now, a new set of lines will be added (from 5 to 10) and from the line number 2 in the following, will be shifted to the end of the file. In addition, the line number 2 will be modified.

```
New file:
line 1
line 5
line 6
line 7
line 8
line 9
line 10
line 2 modified
line 3
line 4
```

The diff tool provides the following information:

- "line 5" is the modification of "line 2".
- "line 6 - line 2 modified" are new lines.
- line 3 and line 4 are old lines and were not modified nor removed.

Thus, the line number 2 and line number 5 are incorrectly detected. Hence, it is necessary to check the percentage of error that it is found when using this method.

As addressed by [Canfora *et al.*, 2009], the modification of the lines are not correctly detected by the diff tool what provokes that analysis based on this approach may face difficulties. In general, it was estimated that only a 13% of the total number of modifications are not correctly detected.

A.3 Bug Fixing Commits detected as Real Fixing Actions

Part of our methodology is based on the fact that some commits are labeled as "bugs". However not all the bugs could have been a bug. Some bugs opened in a bug tracking

system could be there and some will not be fixed, some others are features or some other are simply not a bug

Thus, in order to check this difficulty, some commits have been randomly selected (those whose log information contain the key word bug). This study is based in comm-central project.

List of randomly selected commits fixing a bug in comm-central project:

2038, 5119, 4543, 1027, 5163, 542, 2826, 5053, 5929, 1771, 5074, 2192, 1645, 5928, 4968, 1895, 402, 5900, 5288, 5904, 5374, 4996, 4422, 128, 3753, 5147, 1262, 1412, 3026, 1855, 3574, 1522, 4889, 2227, 5685, 2124, 777, 1574, 3854, 2470, 5366, 1353, 4895, 1768, 4377, 5810, 159, 2021, 272, 3587, 3888, 539, 1673, 2386, 4220, 2508, 2294, 5837, 1059, 2138, 5969, 3479, 1873, 2259, 539, 341, 1884, 63, 3367, 4977, 803, 5247, 4517, 312, 3903, 2952, 5845, 1891, 4296, 3781, 2645, 4677, 3547, 4859, 166, 4628, 4816, 2504, 419, 9, 1097, 1937, 3539, 4105, 4593, 3090, 4850, 5725, 4377

15 of them were detected as not been real bugs. Specifically those with these log messages.

```
(TP) True positive:  78
(FP) False positive:  7
(TN) True negatives:  6
(FN) False negatives:  9
Total commits:  100
```

Therefore we evaluated:

- Positive predictive value: $TP/(TP + FP) = 78/(78 + 7) = 91,7\%$
- Negative predictive value: $TN/(FN + TN) = 40\%$
- Sensitivity $= TP/(TP + FN) = 78/78 + 9 = 89,65\%$
- Specificity $= TN/(FP + TN) = 6/7 + 6 = 46,15\%$

Since the *Precision* actually coincides with the positive predictive, and the *Recall* coincides with the sensitivity, we conclude that $precision = 91,7\%$ and $recall = 89,65\%$.

A.4 Projects where the Source Code was imported but not the History

Generally speaking, several FLOSS projects have to deal with the idea of migrating from one SCM to another. This is the specific case of the Mozilla community where the initial SCM used was CVS, but in 2007 they started to migrate to a distributed SCM such as Mercurial.

The centralized SCM make impossible to be used as a part of this experiment since part of the studies are based on the date provided by the repository which is locally stored depending on the time zone given by an user [1]. However, using a centralized SCM, the field date contains only the date when the commit was submitted to the server storing that centralized date and not the distributed one.

Bringing in context the projects analyzed for this dissertation, the way to check if the history was migrated, consists of checking the first or second commit and its log message found in the Mercurial SCM or looking for the specific and key words "HG_REPO_INITIAL_IMPORT".

From the set of initial selected projects only camino, chatzilla, dom_inspector and schema_validation migrated their history from CVS to Mercurial, in the rest of them, that migration was not done and started the repository from scratch.

A.5 Problems found during the Retrieval Process

Thanks to the metadata stored in the SCM, it is possible to determine when a line was introduced, who introduced it, and its content (needed to identify that it is not a blank line or a comment).

However the huge set of them, makes impossible to analyze one by one.

Next set of problems have been detected, during the approach of following the life of a line as a key points:

- Addition of lines in a given commit and removal of them after some commits.
- Removal of lines in a given commit and addition of the same lines after some commits.
- Addition of a set of lines (merge).
- Deletion of a set of lines (refactoring, modularization).
- Continuous additions and deletions for a set of commits.
- Renaming files
- Movement of files

[1] http://hgbook.red-bean.com/read/a-tour-of-mercurial-the-basics.html

Appendix B

Replicability of the Results

Mining software repositories is a complex task in time, tools and datasets. Different authors have dealt with the issue of replicability in software engineering [Shull *et al.*, 2008], [Basili *et al.*, 1999]. In addition, Robles [Robles, 2010] and later in [González-Barahona & Robles, 2011] have raised a set of questions directly related to this problematic.

In first place, there are several elements that may be of interest for reproducibility:

- **Original data source**: the original data sources used along this dissertation can be found in two different places. In first place, the SCM where the history of the changes is stored can be found at the Mercurial repository of the Mozilla community[1]. And in second place in the bug tracking system where the bug reports are stored. This is found in the Bugzilla of Mozilla[2].

  In addition, in order to have a frozen version of the repository used along this dissertation by means of the *hg log* command, the original software development repositories can be found at the website of this dissertation[3].

- **Retrieval methodology**: the retrieval methodology is based on the use of the *BlameMe*[4], *Cloc*[5] and *Bicho*[6] tools. The tool that was created for specific purposes for this dissertation was BlameMe and the retrieval methodology is detailed in chapter 3, sections 3.2 and 3.3. For the other tools, extra information can be found in their respective websites. In addition, the specific versions used in this dissertation can be found at `http://libresoft.es/~dizquierdo/thesis/tools/`.

- **Raw dataset**: the raw data obtained from the first steps of the methodology is all stored in MySQL databases. Those are available at `http://libresoft.es/`

[1] `http://hg.mozilla.org/`
[2] `https://bugzilla.mozilla.org/`
[3] `http://libresoft.es/~dizquierdo/thesis/raw_repositories/`
[4] `http://git.libresoft.es/blameme/`
[5] `http://cloc.sourceforge.net/`
[6] `http://git.libresoft.es/bicho/`

`~dizquierdo/thesis/databases/`. The tables that are obtained from this initial step are the ones specified at chapter 3, section 3.2 where the tables *scmlog*, *blame* and *files* are created.

- **Extraction methodology**: this methodology is detailed in chapter 3, section 3.4. While the scripts created and needed can be found at .
- **Study parameters**: the initial filter applied over the dataset is detailed in chapter 3, section 4.2.
- **Processed dataset**: the processed dataset is referenced by the tables *bug_seeded_fixed_filtered* and *seeded_fixed_couples*.
- **Analysis methodology**: the tools and scripts used to retrieve the necessary data are found at `http://libresoft.es/~dizquierdo/thesis/tools` and at `http://libresoft.es/~dizquierdo/thesis/scripts/` respectively.
- **Results dataset**: the final datasets are found as charts and also as *csv* files.

In addition, for each of the previous elements, several attributes were detected by [González-Barahona & Robles, 2011] as having an impact in the replicability of the experiments. Their definitions are found in the following set, based on previous literature:

- **Identification**: place where the original data source can be obtained
- **Description**: how detailed is the published information about the element (the elements are those detailed above such as Raw dataset, study parameters or the analysis methodology among others).
- **Availability**: How easy to obtain the element is for a researcher.
- **Persistence**: for how long will be each element in the future
- **Flexibility**: adaptability of the element to new environments

With this respect, the following attributes are expanded:

- **Identification**: the several original places of the datasets used along this dissertation can be found in the repositories already mentioned in the previous list of them. An URL was specified and methodological parts of the thesis referenced when needed.
- **Description**: each of the elements contains a specific description in the methodology chapter.

- **Availability**: the URLs specified in previous lists of elements facilitates the access to the datasets by the researchers. In addition, those are referenced from the website of the thesis.

- **Persistence**: it is expected to have accesible the website as long as the research group, LibreSoft, exists. Thus, this is not a server created *ad hoc*.

- **Flexibility**: the adaptability of the elements to the new environment is detailed in the chapter 5, section 5.1, where the Threats to validity are specified.

Appendix C

Resumen en Castellano

C.1 Introducción

Consideremos un proyecto con varios años de vida y con un grupo específico de desarrolladores donde los roles están bien definidos. El trabajo típico de estas personas será el de trabajar en el apartado de nuevas funcionalidades acorde a los requisitos de los usuarios, realizar cierto mantenimiento del código fuente o procesos de refactorización entre otros.

Ahora bien, el proceso de introducción de errores es un problema inevitable dentro del desarrollo software y cuanto antes se encuentre un error, antes podrá ser eliminado. Es decir, cualquier desarrollador, con cierta probabilidad introducirá nuevos errores en el código fuente.

Dependiendo del momento de ciclo de vida de un proyecto, el arreglo de ese error supondrá cierto tiempo. En el caso de encontrarnos en las primeras fases de desarrollo o incluso en la fase de requisitos, arreglar un error supone un tiempo. Sin embargo, en fases más avanzadas, ese tiempo puede llegar a multiplicarse por 100. Por lo tanto, el encontrar errores en el código fuente con cierta premura es un factor primordial a tener en cuenta y que ayudaría a estimar futuros errores. De hecho, las estimaciones tradicionales indican que alrededor de un 30% del coste de un proyecto se centra en la fase de desarrollo, mientras que el resto, un 70% se centra en la fase de mantenimiento [Pressman & Pressman, 2004].

Centrándonos en la fase de mantenimiento, cuatro tipos diferentes de cambios han sido definidos:

- **Adaptativos**: son los cambios realizados para adaptar el software a nuevos requisitos.
- **Perfectivos**: cambios realizados para mejorar la mantenibilidad del producto.
- **Correctivos**: modificaciones en el código fuente que arreglan errores encontrados.

- **Preventivos**: cambios en el código fuente realizados para prevenir futuros errores.

Focalizando el estudio en los propios desarrolladores, como ya se ha comentado, un desarrollador introducirá con cierta probabilidad un error en el código fuente. Ese error se asume inicialmente que ha sido introducido de manera *involutaria* y que además no ha tenido que ser la misma persona que arregla el error, la que previamente lo introdujo. El proceso de introducción de errores en el código fuente ha sido definido usando la metáfora de plantar una semilla (*to seed* en inglés), aunque otras maneras de definirlo pueden ser usadas, como es el caso de simplemente introducción involuntaria de errores o cambios erróneos.

Teniendo todo esto en cuenta, esta tesis se enfoca en el estudio de la introducción involuntaria de errores en el código fuente, su inclusión como un evento más en el ciclo de vida de los errores y la búsqueda de causas de índole humana que potencien su aparición.

Como factor humano se entienden aquellas métricas que no tienen que ver directamente con el código fuente y que debido a diversas causas dependen únicamente de las personas. Esta tesis, concretamente se enfoca en la relación existente entre experiencia y *calidad* del cambio del código fuente [1].

Como segundo factor humano a estudiar, se ha seleccionado el momento del día como posible causante de errores. Ya se ha mencionado que se tiende cada vez más hacia una distribución del esfuezo alrededor del mundo e incluso alrededor de las 24 horas del día. Es por ello, que este conjunto de experimentos se focalizarán en cómo debido a ciertas presiones externas, existen o no diferencias en cuanto a calidad de los cambios en el código fuente por parte de los desarrolladores en las diferentes horas del día. Como ejemplo, se puede mencionar el hecho de la existencia de fechas límite de entrega que provoquen cierto *estrés* a los desarrolladores. O por ejemplo, que trabajen fuera de su horario habitual, lo cual añadiría cansancio acumulado a lo largo del día a sus últimas modificaciones.

C.2 Antecedentes

Como se ha comentado, los experimentos de esta tesis se pueden dividir en dos grandes grupos: por un lado aquellos relativos al *estudio del ciclo de vida de los errores* y por otro lado aquellos relativos al estudio de las *causas potenciales* que hacen que los desarrolladores los introduzcan. Además, la metodología usada ha sido parcialmente utilizada por otros autores en experimentos similares.

Un resumen de la literatura más importante en la que se basa esta tesis ha sido resumida en la tabla C.1.

[1] Entiéndase calidad como la simplificación binaria de demostrar si un cambio en el código fuente ha conllevado posteriormente la aparición de un error o no

<table>
<tr><th colspan="2">Metodología</th></tr>
<tr><th>Minería de datos</th><th>Cambios erróneos o no</th></tr>
<tr><td>[Robles et al., 2009b],
[Ukkonen, 1985]
[Canfora et al., 2007]
[Chen et al., 2004],
[Śliwerski et al., 2005a],
[Williams & Spacco, 2008],
[Loh & Kim, 2010]
[Canfora et al., 2009]</td><td>[Antoniol et al., 2008],
[Kim et al., 2008c],
[Canfora et al., 2007],
[Kim et al., 2006]</td></tr>
<tr><th colspan="2">Ciclo de vida de los errores</th></tr>
<tr><td colspan="2">[Sommerville, 2006], [Kan, 2002]
[Kagdi et al., 2008], [Sliwerski et al., 2005b]
[Guo et al., 2010], [Capiluppi et al., 2010]
[Hao-Yun Huang & Panchal, 2010], [Pan et al., 2009]
[Gokhale & Mullen, 2010]</td></tr>
<tr><th colspan="2">Factores humanos</th></tr>
<tr><th>Estrucutura organizativa</th><th>Experiencia</th></tr>
<tr><td>[Crowston & Howison, 2005],
[Robles, 2006b]
[Robles et al., 2009a],
[Nagappan et al., 2008]
[Conway, 1968],
[by Andy Oram & Wilson, 2010]</td><td>[Mockus & Herbsleb, 2002],
[Minto & Murphy, 2007],
[McDonald & Ackerman, 2000],
[Terceiro et al., 2010]
[Izquierdo-Cortazar et al., 2009]</td></tr>
<tr><th>Análisis del factor horario</th><td></td></tr>
<tr><td>[Śliwerski et al., 2005a],
[Eyolfson et al., 2011]</td><td></td></tr>
</table>

Table C.1: *Resumen del estado del arte usado en la tesis y dividido por temática*

C.2.1 Antecedentes: metodología

Parte de la metodología está basada en literatura ya existente. El proceso de minería de datos se realiza sobre repositorios de información públicamente accesibles y que contienen información acerca del desarrollo de un producto. Desde diferentes perspectivas, diversos autores han trabajado distintas líneas de investigación [Robles, 2006b], [D'Ambros *et al.*, 2008]. Y esto ha llevado hacia la creación de diversos meta repositorios de información, los cuales proporcionan un punto de entrada a los investigadores [Gonzalez-Barahona *et al.*, 2010]. El uso de este tipo de *almacenes* ayuda a que la investigación evite entrar en los problemas de caracter técnico a los que se enfrenta la minería de datos y directamente obtenga la información que se necesita. Además esto tiene diversas ventajas como es el caso de una homogeneización de datos y su gran cantidad disponible. Entre otros proyectos, merece la pena mencionar a *FLOSSMole* [Howison *et al.*, 2006], *SourceForge Data Archive* [Antwerp & Madey, 2008] o *FLOSSMetrics* [Herraiz *et al.*, 2009].

Sin embargo, a lo largo de esta tesis se han necesitado otro tipo de datos no disponibles en dichos meta repositorios. Esto ha conllevado la creación de herramientas específicas y el uso de otras, entres las cuales se citan las siguientes:

- **BlameMe**[2]: herramienta que guarda en una base de datos relacional las diferencias entre dos versiones del código fuente obteni

- **Bicho**`http://git.libresoft.es`: herramienta que guarda en una base de datos relacional información acerca de los informes de error que se obtienen de un gestor de errores.

- **Cloc**[3]: herramienta que analiza el código fuente y calcula el número de líneas de código fuente y los lenguajes de programación usados.

En el caso de las dos últimas herramientas, éstas ya existían previamente y han sido usadas en diversos contextos. Respecto a la primera de ellas, ha sido necesario crearla desde cero analizando la información que se obtiene a través de la comparación de dos versiones del código fuente. Esto generalmente se realiza usando una herramienta denominada *diff* que permite realizar esa comparación.

Esta herramienta, a su vez, se basa en un algoritmo desarrollo en la década de los 80 y que se explica en [Ukkonen, 1985], [Miller & Myers, 1985] y [Myers, 1986].

Este tipo de herramientas permite el análisis a nivel de línea de un proyecto y mejora el estudio de evolución y esfuerzo de desarrollo [Canfora *et al.*, 2007]:

[2] `http://git.libresoft.es`
[3] `http://cloc.sourceforge.net/`

- **Estudios de evolución**: entre otros se mejora el estudio acerca de la autoría de los cambios en el código fuente, permite estudiar la evolución de los propios cambios en el código fuente (también llamados *parches*) y permite realizar estudios acerca de clonación de código fuente.
- **Estudios de estimación de esfuerzo**: al conocer la autoría exacta de una modificación en el código fuente, esto permite mejorar los actuales modelos de estimación de esfuerzo y costes.

Respecto a la detección de cambios en el código fuente que arreglan o introducen errores, la metodología sigue un proceso similar al de otros artículos. En primer lugar, una de las suposiciones básicas de la tesis sigue la que usa el algoritmo *SZZ* [Śliwerski *et al.*, 2005a]. Éste indica que dado una modificación en el código que arregla un error, las líneas que hayan sido *modificadas* o *eliminadas* son las potenciales causantes de dicho error o al menos son parte del problema. Por lo tanto, basta con retroceder en la vida de dichas líneas en el sistema de control de versiones hasta el momento donde fueron previamente *modificadas* o *añadidas*. Además del artículo ya citado, ésta metodología ha sido usada para identificar justo el problema aquí tratado: cambios en el código fuente que son erróneos o no [Kim *et al.*, 2006], [Kim *et al.*, 2008c].

C.2.2 Antecedentes: experimentos

Como se ha mencionado anteriormente, la tesis se divide en dos grandes bloques de experimentos. Por un lado el estudio del ciclo de vida de los errores donde se introduce el concepto de introducción involuntaria de errores. Y por otro lado factores de índole humana que pueden potenciar dicha introducción de errores.

Respecto al primero de ellos, el ciclo de vida de los errores ha sido ya estudiado con diferentes propósitos. Esta tesis intentará ir un paso más allá e introducir el evento de introducción involuntaria de un error en el código fuente en el estudio del ciclo de vida de los errores. Esto permitirá realizar estudios respecto al tiempo que lleva arreglar un error en el código fuente. Hasta ahora este tipo de estudios como es el caso de modelos de mantenibilidad [Kan, 2002], estudio de las distribuciones de tiempos [Gokhale & Mullen, 2010] o la construcción de modelos de predicción de erratas [Giger *et al.*, 2010], [Panjer, 2007] se han basado única y exclusivamente en el tiempo de arreglo de errores que se obtienen de los sistemas de control de errores. Sin embargo, esta tesis estudiará estos tiempos de arreglo de errores usando como fuente de datos el sistema de control de versiones.

El uso del concepto de introducción involuntaria de errores en el código fuente abre diversas preguntas de investigación que pueden ser ahora respondidas. Entre ellas, esta tesis abarcará la realización de un estudio básico de tiempos de arreglo de errores desde que

estos son introducidos hasta que son descubiertos (abriendo el correspondiente informe de error). Y posteriormente desde que son descubiertos hasta que son arreglados. Esto permitirá realizar comparaciones con estudios ya existentes. Además, estos tiempos permitirán tener información más completa del ciclo de vida de los errores, lo cual redundará en una mejora de los actuales modelos de predicción de aparición de erratas.

Además, comprendiendo como se lleva a cabo el arreglo de errores en el código fuente y el tiempo que suelen llevar, permitirá la realización de modelos que estimen el tipo de distribución que siguen. Es decir, se podrá estudiar cómo de rápido se descubren los errores que han sido involuntariamente introducidos.

Respecto al estudio de diversos factores humanos que potencien la introducción de errores, se han tomado como referencia dos de ellos. Uno más extendido en la literatura como es el caso de la *experiencia* y otro menos extendido como es el caso de la *hora del día* en el que se trabaje. Al ser factores típicamente dependientes del desarrollador y no de factores externos, es necesario hacer mención a la estructura típica seguida en las comunidades de software libre, que es el denominado *modelo de cebolla* [Crowston & Howison, 2005]. Dicho modelo pone de relieve las diferencias en cuanto al esfuerzo dedicado por los diferentes desarrolladores de una comunidad. Se ha hecho patente la existencia de un núcleo central de desarrolladores denominados *core* que suelen desarrollar alrededor del 80% del total de la actividad. Además de que el esfuerzo está claramente balanceado hacia un grupo de personas, existen otros factores denominados *sociales* que modelan la evoluación de estas comunidades. Entre otros cabe destacar la regeneración de desarrolladores [Robles & González-Barahona, 2006], territorialidad del desarrollo [German, 2004b] u otros. Concretamente estos factores *sociales* han sido estudiados como mejores predictores de futuros errores [Nagappan *et al.*, 2008] (cuyo precursor fue la famosa ley de Conway [Conway, 1968]) frente a las métricas típicas de calidad y mantenibilidad del código fuente [Kan, 2002].

En el caso concreto de la experiencia de los equipos de desarrollo, diferentes autores han tratado el tema de obtener métricas cuantitativas al respecto. De entre todos ellos, tres definiciones han sido detectadas como *cuantificables*:

- **Número de cambios en el código fuente**: básicamente mide el número de cambios detectados en el sistema de control de versiones por un desarrollador. A mayor número de cambios, mayor es la experiencia acumulada [Mockus & Herbsleb, 2002].

- **Número de cambios en el código fuente que arreglan errores**: otros autores han escogido un subgrupo del anterior basado en los cambios en el código fuente que arreglan errores [Ahsan *et al.*, 2010].

- **Territorialidad**: finalmente la territorialidad se mide como el número de ficheros

modificados únicamente por un desarrollador [German, 2004a].

Finalmente el estudio de las horas del día como potenciadores de la introducción de errores en el código fuente ha sido previamente tratado por diversos autores. Inicialmente y basado en el estudio del *kernel de Linux* y el proyecto *PostgreSQL* se detectaron que las horas fuera del horario típico de oficina (se puede tomar como ejemplo las horas comprendidas entre las 09:00 y las 18:00) son más propensas a mostrar una mayor concentración relativa de cambios erróneos [Eyolfson *et al.*, 2011]. Esta tesis intentará confirmar o no estos resultados usando como base del estudio la Fundación Mozilla y añadiendo diversos estudios acerca de la distribución general del esfuerzo y la experiencia (éste último punto basado en la división entre desarrolladores *core* y el resto que se realiza en el *modelo de cebolla*).

C.3 Objetivos

A continuación se puede encontrar un listado de los objetivos principales de la tesis:

- **Estudio del ciclo de vida de los errores** en general y del evento de introducción de errores en el código fuente en particular. Este estudio se engloba dentro de los modelos de desarrollo distribuídos. Esta objetivo estudiará cuándo los errores son introducidos de manera involuntaria en el código fuente. Esto permitirá realizar otro tipo de preguntas de investigación como es el caso del estudio del tiempo medio para arreglar un bug y la distribución resultante, estimaciones de ratios de arreglos de bugs o cuántos de los bugs que han sido arreglados han sido posteriormente descubiertos como que han introducido de nuevo un error.

- **Estudio de potenciales factores humanos que potencien la introducción de errores** en el código fuente. Entre otros, se estudiarán de manera específica dos:

  - Factores humanos relacionados con la **experiencia** de los desarrolladores: la experiencia ha sido una métrica que han desarrollado diversos autores para medir cómo de fiable es un desarrollador. En esta tesis se ven tres puntos de vista para medir experiencia: cantidad de modificaciones en el código fuente, modificaciones que arreglan errores y territorialidad del código fuente [4] Estas definiciones de experiencia se correlarán con la *calidad* de los cambios, midiéndose desde el punto de vista de si éstos serán erróneos o no.

[4]Esta métrica es obtenida a través del número total de ficheros que sólo un desarrollador toca sin que nadie más lo haya hecho. Es decir, serán todos aquellos ficheros cuya autoría sea para un único desarrollador.

– Factores humanos relacionados con la **hora del día** en la que se realiza la modificación del código fuente: la elección de la hora del día como posible factor procede del estudio observacional del tipo de actividad llevada a cabo por los desarrolladores. La actividad agregada en periodos de 24 horasofrece como resultado dos picos de actividad. Uno realizado en un horario típico de oficina (entre las 09:00 y las 18:00) y otro realizado a partir de las 19:00 horas hasta medianoche. Como es sabido, además, en las comunidades de software libre es típico encontrar grupos de desarrolladores *híbridos* que se pueden dividir entre aquellos que reciben dinero por su trabajo y los voluntarios. Además, cualquier desarrollador tendrá un horario tipo donde realizará casi todo trabajo. Sin embargo, existen diversas causas que pueden forzar a los desarrolladores a trabajar en horarios significativamente diferentes de su horario tipo. Por lo tanto se estudiará la relación existente entre la hora del día y la *calidad* del tipo de modificación llevada a cabo desde diversos puntos de vista.

C.4 Metodología

La metodología general usada se basa en el estudio del sistema de control de versiones y en algunos casos añadir información extra del sistema de control de errores. Como visión global, esta tesis se enfoca en el estudio de los cambios en el código fuente que han arreglado un error y que gracias a la existencia del sistema de control de versiones, permite trazar ciertos parámetros que son útiles para llegar hasta las raíces de ese error.

Para ello, la metodología general se divide en tres grandes pasos:

1. Detección de modificaciones en el código fuente que arreglan errores:
2. Identificación de las líneas que formaron parte de esa modificación:
3. Detección de las modificaciones en el código fuente donde se introdujo el error:

C.4.1 Detección de cambios que arreglan errores

Cualquier desarrollador puede realizar cambios en el código fuente, siempre que tenga los permisos pertinentes. Con el advenimiento de los sistemas de control de versiones distribuídos, como es el caso de Git o Mercurial, el tener permisos para realizar modifications quedó diluído para dar paso a una manera más abierta de colaboración en las comunidades.

Esta tesis se basa en el estudio del sistema de control de versiones denominado Mercurial, el cual proporciona información específica. Un ejemplo de un log de Mercurial puede verse a continuación:

```
$ hg log -r11106
changeset:   11106:ce7b4d1c9978
user:        mozilla.mano@sent.com
date:        Thu Jan 31 10:05:22 2008 -0800
summary:     Bug 411088 - when deleting a tagged bookmark from the places organizer,
             the tag remains. r=dietrich, a=beltzner.
```

Este log ofrece información como:

- changeset: proporciona información acerca del "hash" value que identifica el cambio en el código fuente frente al resto.

- user: proporciona información acerca de quién realizó el cambio. En algunos casos, además del nombre y apellido del desarrollador, la dirección de correo puede también ofrecerse.

- date: este campo proporciona información acerca de cuándo específicamente se realizó la modificación en el código fuente. Como se puede comprobar, ofrece información acerca de la zona horaria desde la que se realizó (-08:00) , así como de la fecha concreta. Esto proporciona información valiosa frente a otros tipos de sistemas de control de versiones más antiguos, como es el caso de la hora local del desarrollador (junto con su zona horaria) y no la hora centralizada del servidor, como es el caso de otros sistemas de control de versiones centralizados, como CVS o Subversion.

- summary: este campo proporciona información que ha dejado el desarrollador correspondiente. Dependiendo de la comunidad, este campo se rellena de una manera estandárizada o por el contrario, se deja a criterio del desarrollador el rellenarlo según le parezca. En el caso de la comunidad Mozilla, este tipo de cambios está regulado y en el caso de que un error haya sido arreglado, se indica de manera precisa usando en primer lugar la palabra clave "Bug" o "bug" y a continuación un entero que referencia al número de informe de error en el sistema de gestión de errores.

Por lo tanto, las modificaciones en el código fuente que arreglan un error son detectadas a través del estudio del mensaje dejado por los desarrolladores. Concretamente usando el heurístico especificado anteriormente. Además se ha realizado una validación de dicho heurístico que ha mostrado una precisión de un 91.7% y un recall de 89.65%.

C.4.2 Detección de las líneas que tomaron parte

Además de la información obtenida a través del comando "hg log", existen otros comandos que ofrecen información extra acerca de una modificación en el código fuente. Concretamente el comando "hg diff" ofrece información acerca de las diferencias entre dos versiones para cada uno de los ficheros que tomaron parte. Un ejemplo puede verse a continuación:

```
$ hg diff -r11106 -r11107
diff -r ce7b4d1c9978 -r cbbdb92a2c31 browser/components/places/content/controller.js
--- a/browser/components/places/content/controller.js
Thu Jan 31 10:05:22 2008 -0800
+++ b/browser/components/places/content/controller.js
Thu Jan 31 10:10:40 2008 -0800
@@ -276,10 +276,14 @@
       if (nodes[i] == root)
         return false;

+      // Disallow removing shortcuts from the left pane
+      var nodeItemId = nodes[i].itemId;
+      if (PlacesUtils.annotations
+                     .itemHasAnnotation(nodeItemId, ORGANIZER_QUERY_ANNO))
+        return false;
+
       // Disallow removing the toolbar, menu and unfiled-bookmarks folders
-      var nodeItemId = nodes[i].itemId;
       if (!aIsMoveCommand &&
-          PlacesUtils.nodeIsFolder(nodes[i]) &&
           (nodeItemId == PlacesUtils.toolbarFolderId ||
            nodeItemId == PlacesUtils.unfiledBookmarksFolderId ||
            nodeItemId == PlacesUtils.bookmarksMenuFolderId))
```

Como ya se ha comentado, el comando "hg log" ofrece información que nos permite identificar que un commit ha arreglado un error y el comando "hg diff" proporciona información específica acerca de las líneas que han sido añadidas, modificadas o eliminadas y sus ficheros asociados. En el ejemplo que nos ocupa, sólo hay un fichero que haya sido modificado: "controller.js". El dato que se encuentra entre las dos arrobas: "@@ -276,10 +276,14 @@" indica las posiciones del fichero original que aparecen y las del fichero final.

Finalmente, las líneas que aparecen a continuación son las líneas que han tomado parte en este cambio. En el caso de que una línea comience con el caracter "+", indica que esa línea ha sido añadida. Si la línea comienza con el caracter "-", indica que la línea ha sido eliminada. En el caso de que se encuentren cierto número de líneas comenzando por "+" y consecutivamente cierto número de líneas que comiencen por el caracter "-", esas líneas se consideran como modificadas.

En el ejemplo que nos ocupa, 6 líneas han sido añadidas y 2 eliminadas. El resto de líneas que comienzan por un espacio en blanco simplemente son líneas que aún permanecen en el fichero y no han sido afectadas por los cambios.

C.4.3 Detección de los orígenes del error

Por lo tanto, hasta ahora se conocen las modificaciones que tomaron parte de un arreglo de un error y por otro lado se conocen las líneas que tomaron parte en dicha modificación. Para detectar los orígenes del error, se usa como base el algoritmo "SZZ" [Śliwerski *et al.*, 2005a] el cual determina que aquellas líneas que en una modificación del código fuente que haya arreglado un error y que tengan el estado de eliminadas o modificadas, son las causantes de dicho error. Es por ello, que para detectar el origen del error, hay que trazar dichas líneas en la historia hacia atrás.

La metodología propuesta, lo que realiza es el guardado de todas las diferencias entre cada par de versiones, gracias a la herramienta desarrollada expresamente denominada BlameMe. Dicha herramienta, guarda toda la información asociada y que ha sido mencionada anteriormente. Cada línea tiene un identificativo único y está asociada a su vez a un fichero con un identificativo único. Esto nos permite, para una línea en concreto, para un fichero en concreto, en un momento de la historia en concreto, descubrir qué líneas fueron eliminadas y modificadas y trazar su historia.

C.4.4 Herramientas

Como ya ha sido comentado, las herramientas usadas en el transcurso de la tesis son tres:

- BlameMe: esta herramienta se encarga de guardar las diferencias encontradas entre cada par de versiones dado un sistema de control de versiones.

- Bicho: esta herramienta, dado una URL, guarda toda la información asociada a un informe de error en un bug tracking system.

- Cloc: esta herramienta cuenta las líneas de un proyecto y ofrece información acerca del tipo de lenguaje de programación, ficheros totales y otras métricas básicas.

C.4.5 Métricas consideradas

Las métricas que se han usado para cada uno de los diferentes experimentos se resumen en la siguiente lista (algunos de ellos se encuentran en su nombre original en inglés debido a que recogen en mayor medida el significado de la métrica en sí misma).

1. **Ciclo de vida de los errores**:
   (a) Actividad general: medida en número de modificaciones en el sistema gestor de versiones
   (b) Bug fixing commits: o subconjunto de la actividad general. Éstos son las modificaciones que han sido detectadas solventando algún error
   (c) Bug seeding commits: o subconjunto de la actividad general. Éstos son las modificaciones que han sido detectadas introduciendo algún error en el código fuente
   (d) Desarrollador: o author de las modificaciones asociadas a un *commit*
   (e) Hora de *commit*: momento en el que se realiza un *commit*. Se calcula hasta el nivel de segundos
   (f) Ficheros: ficheros que han sido modificados en un *commit*
   (g) Líneas de código fuente: líneas de código fuente modificadas
   (h) Lenguaje de programación: asociado a un fichero según su extensión
   (i) Tiempo de arreglo de un error en el sistema de versiones: tiempo total que transcurre desde que el error ha sido involuntariamente introducido en el código fuente hasta que se arregla
   (j) Apertura de informe de error: su número único identificativo en el sistema gestor de erratas está relacionada con el número que indica un desarrollador a la hora de arreglar un error en el sistema de versiones
   (k) Cerrado de informe de error: último paso en la vida de un error en el sistema gestor de errores (podría darse el caso que éste fuera abierto de nuevo)
   (l) Tiempo de arreglo de un error en el sistema gestor de errores: tiempo que transcurre desde que se abre un informe de error hasta que se cierra
2. **Factores humanos: experiencia**
   (a) Actividad general
   (b) Bug seeding commits
   (c) Bug fixing commits

(d) Desarrollador

(e) Proporción de modificaciones erróneas: se define como el porcentaje resultante entre el número de modificaciones detectadas como erróneas (*bug seeding commits*) y la actividad total en un periodo de tiempo

(f) Coeficiente BMI : se define como el porcentaje resultante entre el número de modificaciones al código fuente que arreglan un error y las modificaciones que son detectadas como erróneas en un periodo de tiempo

(g) Territorialidad: se define como el porcentaje de ficheros que han sido modificaciones únicamente por un solo desarrollador

(h) Diferencia en tipos de actividad: se define como la diferencia entre toda la actividad agregada de aquellas modificaciones que arreglan un error y aquellas modificaciones que introducen errores

3. **Factores humanos: hora de actividad**

(a) Actividad general

(b) Bug seeding commits

(c) Bug fixing commits

(d) Proporción de modificaciones erróneas

(e) Desarrollador

(f) Desarrolladores principales y no principales: los desarrolladores principales se definen como aquellos que usualmente llevan a cabo el 80% del total de modificaciones en el sistema

(g) Hora de *commit*: momento en el día en el que se realiza una modificación en el código fuente

C.5 Resultados

En general los resultados se pueden dividir entre los tres objetivos principales de la tesis: ciclo de vida de los errores, errores humanos debidos a la experiencia y errores debidos a la hora del día de trabajo. Para cada uno de ellos, los resultados más importantes serán citados.

Resultados: Ciclo de vida de los errores

El estudio del ciclo de vida de los errores, añadiendo el momento en el que éstos han sido introducidos en el código fuente ha permitido realizar diversas preguntas de investigación cuyos resultados pueden ser resumidos en las siguientes lecciones aprendidas:

- Estudio del tiempo de arreglo de un error:
  - Lección aprendida: las distribuciones del tiempo de arreglo de un error en el código fuente no siguen ni una distribución de tipo normal ni una distribución de tipo log-normal. Esto es extensible a las distribuciones estudiadas en el sistema de control de errores. Además, las comparaciones entre las distribuciones del tiempo de arreglo de un error en el sistema de control de versiones y de errores indican que siguen distribuciones de datos diferentes.
  - Lección aprendida: la definición de políticas referentes a la importancia de ciertos proyectos frente a otros indica que ayudan a reducir el tiempo medio de arreglo de un bug tanto en el código fuente como en el sistema de control de errores. Por lo tanto, la definición de políticas de aseguramiento de la calidad en la fundación Mozilla ayuda al proceso de mantenimiento software.
- Lección aprendida: el ratio de arreglo de errores (o dicho de otra manera, la rapidez con la que los errores involuntariamente introducidos en el código fuente son encontrados) suele seguir una distribución decreciente lineal o exponencial. En todos los casos, excepto en el caso del proyecto Mozilla Central, la mayor parte de los datos siguen un decrecimiento lineal.
- Lección aprendida: el porcentaje de cambios en el código fuente que arreglan un error y que posteriormente son detectados como que han introducido un nuevo error, varía entre un 27% en el caso del proyecto Camino hasta un 87% en el caso del proyecto Comm Central. Además, en este caso aquellos proyectos declarados como importantes para la Fundación Mozilla son los que presentan un ratio más alto de arreglos que posteriormente son detectados como que han introducido nuevos errores.

Resultados: Factores humanos - experiencia

Respecto a los factores humanos, la experiencia ha sido medida de tres formas diferentes, según ha sido estudiado en la literatura desde un punto de vista cuantitativo: número de modificaciones en el código fuente, número de modificaciones en el código fuente que arreglan un error y autoría de los cambios (usando como métrica el número de ficheros que son modificados por un único desarrollador). Dichas métricas han sido posteriormente correladas con el ratio de introducción de errores y entre ellas mismas.

- Lección aprendida: no existe empiricamente hablando, relación entre cualquier tipo de manera de medir la experiencia y el ratio de introducción de errores. Es decir, la experiencia, al contrario de lo esperado, no hace que un desarrollador introduzca menos errores ni existe un decrecimiento en su actividad a lo largo de su desarrollo profesional en el proyecto.

- Lección aprendida: Además, entre las propias métricas usadas para medir experiencia de los desarrolladores, no existe evidencia empírica que demuestre relación entre ellas, excepto en el caso del número de modificaciones en el código fuente y número de modificaciones en el código fuente que arreglan un error. Este último resultado indica que aquellos desarrolladores que tienen una actividad creciente, también muestran una actividad creciente cuando arreglan errores.

- Lección aprendida: el estudio del tipo de actividad de un desarrollador, permite realizar una caracterización de desarrolladores donde uno de los extremos es el de mantenedor "puro" y el otro el de desarrollador que nunca ha arreglado un error en el código fuente (o al menos lo ha indicado en el sistema de control de versiones).

Resultados: Factores humanos - hora del día

En el caso de los factores humanos relativos a la hora del día, las siguientes lecciones han sido aprendidas:

- Lección aprendida: las distribuciones agregadas en periodos de 24 horas y anualizadas indican que son diferentes. Esto indica que las comunidades evolucionan y los desarrolladores no tienden a trabajar en un horario fijo. Además las distribuciones de modificaciones en el código fuente son diferentes entre los distintos proyectos, lo cual impide agregarlos para posteriormente estudiarlos.

- Lección aprendida: los picos más altos de ratio de introducción de errores en el código fuente suelen producirse durante el segundo periodo de actividad o durante la madrugada.

- Lección aprendida: no existen diferencias empíricas entre el ratio de introducción de errores de los desarrolladores que llevan a cabo el 80% del total del trabajo, frente al resto de los desarrolladores del proyecto. En otras palabras: si se usa el número de commits como medida para cuantificar experiencia y se realiza una división entre los desarrolladores más importantes frente al resto de la comunidad, no existen diferencias sustanciales comparando el ratio de introducción de errores.

- Lección aprendida: los cambios en el código fuente, fuera del horario tipo de un desarrollador son más propensos a tener errores.

C.5.1 Observaciones sobre la Fundación Mozilla

Como ya se ha comentado, la Fundación Mozilla ha desarrollado una serie de políticas para el proceso de detección y solución rápida de los errores que aparezcan en el código fuente. Para ello, recursos específicos como es el caso del tiempo de desarrolladores con más

experiencia, son usados para realizar una doble comprobación del código fuente cuando se realice un cambio.

Sin embargo, se han detectado ciertos comportamientos que parecen indicar una mayor introducción relativa de errores. Concretamente, se han detectado grandes diferencias entre los diferentes proyectos en el ratio de modificaciones en el código fuente que solventan errores, pero que posteriormente han sido detectadas como erróneas. Una posible explicación a este hecho es el continuo desarrollo de ciertas comunidades frente a otras. Sin embargo, por cuestiones de mantenimiento, los ratios de introducción de errores en muchos casos sobrepasaban al número de arreglos que se realizaban. Por lo tanto, cuando esfuerzo extra debería derivarse hacia este tipo de modificaciones en ciertos proyectos, como es el caso de Comm Central que tienen una tendencia mayor a introducir errores.

En el caso de los análisis de modifiaciones a ciertas horas del día, es recomendable realizar una comprobación extra por parte de los desarrolladores en aquellas que se realicen a altas horas de la noche, o durante el periodo que lleva de las 19:00 horas a medianoche.

C.6 Conclusiones y trabajo futuro

La consideración de las estructuras organizativas y factores humanos como potenciales fuentes de errores han arrojado resultados interesantes que pudieran ser tenidos en cuenta a la hora de gestionar equipos de desarrollo.

El uso de la comunidad Mozilla, con políticas específicas en cuanto al mantenimiento del código fuente (asignación de más recursos humanos a la hora de evaluar las modificaciones en el código fuente a los proyectos declarados como principales) ha ayudado a proporcionar datos acerca de cómo se desarolla el código fuente de dichos proyectos. Respecto a los objetivos iniciales del estudio, éstos se dividieron en dos grupos diferenciados: en primer lugar el estudio del ciclo de vida de los errores y en segundo lugar, el estudio de factores humanos que sean más propensos a introducirlos.

Respecto al primero de ellos, el estudio del ciclo de vida de los errores, esta tesis ha analizado los repositorios de código fuente llegando hasta sus orígenes. Entre otros, se ha descubierto como del total de la vida de un error, desde un 60% hasta un 90% de su vida transcurre sin que éste haya sido descubierto. A continuación, éste pasaría al sistema de gestión de errores a través de la apertura de un informe de error. Además, el estudio de la evolución de dichos errores y cómo de rápido son descubiertos ha permitido averigurar que se ajustan en mayor medida a un modelo lineal, lo cual permite mejorar las estimaciones.

Además, el estudio de las modificaciones en el código fuente que arreglan un error, arroja resultados acerca de su proporción de introducción de nuevos errores: estos datos se encuentran en un rango entre un 27% hasta un 93%. Como ejemplo, el último dato indica que en algunos casos, casi un 100% de las modificaciones que arreglan un error

introduce uno nuevo no detectado.

Finalmente, la caracterización de desarrolladores basada en su actividad (dividida entre modificaciones al código fuente que introducen errores y aquellas que arreglan errores), permite realizar una división para determinar los mantenedores de código *puros* que son aquellos que se dedican única y exclusivamente a arreglar errores y no a añadir nuevas funcionalidades al sistema.

Respecto a los factores humanos, el estudio de la experiencia de un desarrollador desde diferentes puntos de vista y la proporción de introducción de errores revela que no existe, al menos en la comunidad Mozilla, una relación directa entre estas dos variables. Sin embargo, el estudio de la hora del día como potencial factor para introducir más errores, ha revelado una mayor proporción de introducción de errores de código fuente en horarios fuera de las típicas *horas de oficina*.

Como conclusión general de la tesis: los objetivos principales han sido conseguidos a través del estudio de las diferentes preguntas de investigación propuestas. En primer lugar, usando el evento de introducción involuntaria de errores en el código fuente (*bug seeding commit)* y en segundo lugar a través del estudio de factores humanos que potencien la introducción de errores como la experiencia y la hora del día en la que suelen trabajar los desarrolladores.

C.6.1 Trabajo futuro

Esta tesis ha realizado un estudio acerca del ciclo de vida de los bugs donde se ha introducido el concepto de introducción de errores en el código fuente y han sido estudiadas las potenciales causas para su aparición.

Focalizando en el caso concreto en el ciclo de vida de los bugs, los datos muestran como para la Fundación Mozilla, el uso de políticas específicas ayuda en la fase de mantenimiento, donde los errores, de media, son solucionados rápidamente. Sin embargo, estos datos deberían ser comprobados en otras comunidades de software libre o en proyectos propietarios donde políticas similares se apliquen. Es por ello, que añadir nuevos proyectos o comunidades al estudio enriquecería los resultados finales.

En el caso de la evolución de los errores, se han obtenido estimaciones acerca del tipo de distribución que siguen los errores que se solucionan. Sin embargo, este tipo de planteamiento no ha sido aún utilizado, lo cual permitiría o ayudaría en la estimación final de costes, basándose en el estudio previo de comportamiento de las comunidades.

Respecto al estudio de potenciales causas de introducción de errores, se ha demostrado como la experiencia, medida desde un punto de vista cuantitativo, parece no ser un buen indicador para medir calidad en el código fuente. Sin embargo, nuevos estudios relacionados con métricas "sociales" a este respecto deberían llevarse a cabo. Entre otros, la apliación de cualquiera de las métricas propuesta por [Nagappan *et al.*, 2008] sería un

buen paso inicial.

Finalmente, respecto a la metodología usada, existen ciertas limitaciones relacionadas con complejidad en tiempo, que impiden la mejora del algoritmo y las suposiciones realizadas al principio de la tesis. Una posible mejora es el uso del algoritmo *ldiff* para determinar mejor qué líneas fueron añadidas, eliminadas o modificadas [Canfora *et al.*, 2009]. Con los datos actuales y los datos ofrecidos por los autores en dicho artículo, el tiempo de cálculo en algunos casos, donde se han detectado modificaciones de cientos de miles de líneas, se dispararía hasta los días o semanas de cálculo para un solo cambio en el código fuente. La mejora de rendimiento de dicho algoritmo, unido al uso de la herramienta BlameMe, mejoraría mucho más la precisión de los datos usados.

Bibliography

[Ackerman & Halverson, 1998] Mark S. Ackerman and Christine Halverson. Considering an organization's memory. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 39–48, New York, NY, USA, 1998. ACM.

[Ahsan *et al.*, 2010] S.N. Ahsan, M.T. Afzal, S. Zaman, C. Gütel, and F. Wotawa. Mining effort data from the oss repository of developer's bug fix activity. *Journal of Information Technology in Asia*, 3:67 – 80, 2010.

[Antoniol *et al.*, 2008] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA, 2008. ACM.

[Antwerp & Madey, 2008] Matthew Van Antwerp and Greg Madey. Advances in the sourceforge research data archive. In *WoPDaSD*, 2008.

[Bachmann *et al.*, 2010] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 97–106, New York, NY, USA, 2010. ACM.

[Bacon, 2009] Jono Bacon. *The Art of Community - Building the New Age of Participation*. O'Reilly, 2009.

[Basili & Perricone, 1984] Victor Basili and Barry T Perricone. Software errors and complexity an empirical investigation. *Communications of the ACM*, 27:42–52, 1984.

[Basili *et al.*, 1999] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Trans. Softw. Eng.*, 25:456–473, July 1999.

[Bird *et al.*, 2009a] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in

bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[Bird *et al.*, 2009b] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[Biyani & Santhanam, 1998] S. Biyani and P. Santhanam. Exploring defect data from development and customer usage on software modules over multiple releases. *Software Reliability Engineering, International Symposium on*, 0:316, 1998.

[Brooks, 1995] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[by Andy Oram & Wilson, 2010] Edited by Andy Oram and Greg Wilson. *Making Software: What Really Works, and Why We Believe It.* O'Reilly, October 2010.

[Canfora *et al.*, 2007] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA, 2007. IEEE Computer Society.

[Canfora *et al.*, 2009] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Ldiff: An enhanced line differencing tool. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 595–598, Washington, DC, USA, 2009. IEEE Computer Society.

[Capiluppi *et al.*, 2007] Andrea Capiluppi, Jesús M. González-Barahona, Israel Herraiz, and Gregorio Robles. Adapting the "staged model for software evolution" to free/libre/open source software. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 79–82, New York, NY, USA, 2007. ACM.

[Capiluppi *et al.*, 2010] A. Capiluppi, A. Baravalle, and N.H. Heap. Open standards and e-learning: the role of open source software. In *Proc. of the 6th International Conference on Open Source Systems (OSS 2010)*, Notre Dame, IN, USA, May-June 2010.

[Chen *et al.*, 2004] Kai Chen, Stephen R. Schach, Liguo Yu, Jeff Offutt, and Gillian Z. Heller. Open-source change logs. *Empirical Software Engineering*, 9:197–210, 2004.

[Christenson & Huang, 1996] Dennis A. Christenson and Steel T. Huang. Estimating the fault content of software using the fix-on-fix model. *Bell Labs Technical Journal*, 1(1):130–137, 1996.

[Conway, 1968] M.E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[Crowston & Howison, 2005] Kevin Crowston and James Howison. The social structure of free and open source software development. *First Monday*, 10(2), 2005.

[D'Ambros *et al.*, 2008] M D'Ambros, H C Gall, M Lanza, and M Pinzger. Analyzing software repositories to understand software evolution. In T Mens and S Demeyer, editors, *Software Evolution*, pages 37–67. Springer, Heidelberg, Germany, 2008.

[De Marco & Lister, 1999] Tom De Marco and Timothy Lister. *Peopleware : Productive Projects and Teams, 2nd Ed.* Dorset House Publishing Company, Incorporated, 1999.

[Denaro & Pezze, 2002] Giovanni Denaro and Mauro Pezze. An empirical evaluation of fault-proneness models. In *Proceedings of the Int'l Conf. on Software Engineering*, pages 241–251. ACM, 2002.

[Ebert & Neve, 2001] Christof Ebert and Philip De Neve. Surviving global software development. *IEEE Software*, 18:62–69, 2001.

[Eick *et al.*, 1992] Stephen G. Eick, Clive R. Loader, M. David Long, Lawrence G. Votta, and Scott Vander Wiel. Estimating software fault content before coding. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 59–65, New York, NY, USA, 1992. ACM.

[Endres & Rombach, 2003] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Addison Wesley, illustrated edition edition, May 2003.

[Eyolfson *et al.*, 2011] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 153–162, New York, NY, USA, 2011. ACM.

[Fernandez-Ramil *et al.*, 2008]
Juan Fernandez-Ramil, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi. Empirical studies of open source evolution. In Tom Mens and Serge Demeyer, editors, *Software Evolution: State-of-the-art and research advances*, chapter 11, pages 263–288. Springer Verlag, 2008.

[Fogel, 2005] Karl Fogel. *Producing Open Source Software : How to Run a Successful Free Software Project*. O'Reilly Media, Inc., 2005.

[Fritz *et al.*, 2007a] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 341–350, New York, NY, USA, 2007. ACM.

[Fritz *et al.*, 2007b] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 341–350, New York, NY, USA, 2007. ACM.

[German, 2004a] Daniel M. German. Using software trails to reconstruct the evolution of software: Research articles. *J. Softw. Maint. Evol.*, 16:367–384, November 2004.

[German, 2004b] Daniel M. German. Using software trails to reconstruct the evolution of software: Research articles. *J. Softw. Maint. Evol.*, 16(6):367–384, 2004.

[Giger *et al.*, 2010] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56, New York, NY, USA, 2010. ACM.

[Girba *et al.*, 2005] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stephane Ducasse. How developers drive software evolution. *Principles of Software Evolution, International Workshop on*, 0:113–122, 2005.

[Gokhale & Mullen, 2010] Swapna Gokhale and Robert Mullen. A multiplicative model of software defect repair times. *Empirical Software Engineering*, 15:296–319, 2010. 10.1007/s10664-009-9115-y.

[González-Barahona & Robles, 2011] Jesús González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, pages 1–15, 2011.

[González-Barahona *et al.*, 2008] Jesús M. González-Barahona, Gregorio Robles, Roberto Andradas-Izquierdo, and Rishab Aiyer Ghosh. Geographic origin of libre software developers. *Information Economics and Policy*, 20(4):356–363, 2008.

[Gonzalez-Barahona *et al.*, 2010] Jesus M. Gonzalez-Barahona, Megan Squire, and Daniel Izquierdo-Cortazar. Repositories with public data about software development. *International Journal of Open Source Software and Processes (IJOSSP)*, 2(2):1–13, 03/2010 2010.

[Graves *et al.*, 2000] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.

[Gregorio Robles & Herraiz, 2011] Daniel Izquierdo-Cortazar Gregorio Robles, Jesus M. Gonzalez-Barahona and Israel Herraiz. *Tools and Datasets for Mining Libre Software Repositories*, volume 1, chapter 2, page 24–42. IGI Global, Hershey, PA, 2011.

[Guo *et al.*, 2010] Philip J. Guo, Thomas Zimmermann, Nachiappann, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *ICSE (1)*, pages 495–504, 2010.

[Hao-Yun Huang & Panchal, 2010] Qize Le Hao-Yun Huang and Jitesh H. Panchal. Analysis of the structure and evolution of an open-source community. In *Proceedings of the ASME 2010 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2010*, 2010.

[Hatton, 1997] Leslie Hatton. Re-examining the fault density - component size connection. *IEEE Software*, 14(2):89–97, 1997.

[Herbsleb & Mockus, 2003] James D. Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29:481–494, 2003.

[Herraiz *et al.*, 2009] Israel Herraiz, Daniel Izquierdo-Cortazar, Francisco Rivas-Hernandez, Jesus M. Gonzalez-Barahona, Gregorio Robles, Santiago Dueñas, Carlos Garcia-Campos, Juan Francisco Gato, and Liliana Tovar. Flossmetrics: Free / libre / open source software metrics. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 281–284, Kaiserlauten, Germany, 03/2009 2009. IEEE Computer Society, IEEE Computer Society.

[Howison *et al.*, 2006] James Howison, Megan Conklin, and Kevin Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering*, 1:17–26, 07/2006 2006.

[Hutchins *et al.*, 1994] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test

adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Izquierdo-Cortazar *et al.*, 2009] Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega, and Jesus M. Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS-42)*, Big Island, Hawaii, USA, January 2009.

[Izquierdo-Cortazar *et al.*, 2012] Daniel Izquierdo-Cortazar, Andrea Capiluppi, and Jesus M. Gonzalez-Barahona. (accepted for publication) are developers fixing their own bugs?. tracing bug-fixing and bug-seeding committers. *International Journal of Open Source Software and Processes (IJOSSP)*, 2012.

[Jiménez *et al.*, 2009] Miguel Jiménez, Mario Piattini, and Aurora Vizcaíno. Challenges and improvements in distributed software development: a systematic review. *Adv. Soft. Eng.*, 2009:3:1–3:16, January 2009.

[Kagdi *et al.*, 2008] Huzefa H. Kagdi, Maen Hammad, and Jonathan I. Maletic. Who can help me with this source code change? In *ICSM*, pages 157–166, 2008.

[Kan, 2002] Stephen H. Kan. *Metrics and Models in Software Quality Engineering (2nd Edition)*. Addison-Wesley Professional, September 2002.

[Khoshgoftaar *et al.*, 1996] T.M. Khoshgoftaar, E.B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. *Software Reliability Engineering, International Symposium on*, 0:364, 1996.

[Kim *et al.*, 2006] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. *Automated Software Engineering, International Conference on*, 0:81–90, 2006.

[Kim *et al.*, 2008a] Sunghun Kim, Jr. E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March/April 2008.

[Kim *et al.*, 2008b] Sunghun Kim, E. James Whitehead Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, 2008.

[Kim *et al.*, 2008c] Sunghun Kim, E. James Whitehead Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, 2008.

[Kim *et al.*, 2008d] Sunghun Kim, E.J. Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181 –196, march-april 2008.

[Kitchenham *et al.*, 2002] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734, August 2002.

[Koch & Schneider, 2002] Stefan Koch and Georg Schneider. Effort, co-operation and co-ordination in an open source software project: Gnome. *Information Systems Journal*, 12(1):27–42, 2002.

[Lehman & Ramil, 2006] Meir Lehman and Juan C. Fernández Ramil. *Software Evolution*, chapter Software Evolution, pages 07–40. Wiley, 2006.

[Lehman, 1979 1980] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213 – 221, 1979-1980.

[Loh & Kim, 2010] Alex Loh and Miryung Kim. Lsdiff: a program differencing tool to identify systematic structural differences. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 263–266, New York, NY, USA, 2010. ACM.

[MacKenzie *et al.*, 2002] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files with GNU diff and patch.* Network Theory Ltd., 2002.

[McDonald & Ackerman, 2000] David W. McDonald and Mark S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, pages 231–240, New York, NY, USA, 2000. ACM.

[Miller & Myers, 1985] Webb Miller and Eugene W. Myers. A file comparison program. *Software - Practice and Experience*, 15(11):1025–1040, 1985.

[Minto & Murphy, 2007] Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 5–, Washington, DC, USA, 2007. IEEE Computer Society.

[Mockus & Herbsleb, 2002] A. Mockus and J.D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 503 –512, may 2002.

[Mockus *et al.*, 2002] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[Myers, 1986] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[Nagappan & Ball, 2005] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292, May 2005.

[Nagappan *et al.*, 2006] Nachiappan Nagappan, Thomas Ball, and Brendan Murphy. Using historical in-process and product metrics for early estimation of software failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 62–74, Washington, DC, USA, 2006. IEEE Computer Society.

[Nagappan *et al.*, 2008] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.

[Nguyen *et al.*, 2010] Thanh H.D. Nguyen, Bram Adams, and Ahmed E. Hassan. A case study of bias in bug-fix datasets. *Reverse Engineering, Working Conference on*, 0:259–268, 2010.

[Pan *et al.*, 2009] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, 2009.

[Panjer, 2007] Lucas D. Panjer. Predicting eclipse bug lifetimes. *Mining Software Repositories, International Workshop on*, 0:29, 2007.

[Podgurski & Clarke, 1990] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16:965–979, 1990.

[Pressman & Pressman, 2004] Roger Pressman and Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 6 edition, April 2004.

[Ramesh *et al.*, 2006] Balasubramaniam Ramesh, Lan Cao, Kannan Mohan, and Peng Xu. Can distributed software development be agile? *Commun. ACM*, 49:41–46, October 2006.

[Robles & González-Barahona, 2006] Gregorio Robles and Jesús M. González-Barahona. Contributor turnover in libre software projects. In Ernesto Damiani, Brian Fitzgerald, Walt Scacchi, Marco Scotto, and Giancarlo Succi, editors, *OSS*, volume 203 of *IFIP*, pages 273–286. Springer, 2006.

[Robles *et al.*, 2006] Gregorio Robles, Jesús M. González-Barahona, and Juan Julián Merelo Guervós. Beyond source code: The importance of other artifacts in software development (a case study). *Journal of Systems and Software*, 79(9):1233–1248, 2006.

[Robles *et al.*, 2009a] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Israel Herraiz. Evolution of the core team of developers in libre software projects. *Mining Software Repositories, International Workshop on*, 0:167–170, 2009.

[Robles *et al.*, 2009b] Gregorio Robles, Jesus M. Gonzalez-Barahona, Daniel Izquierdo-Cortazar, and Israel Herraiz. Tools for the study of the usual data sources found in libre software projects. *International Journal of Open Source Software and Processes (IJOSSP)*, 1(1):24–45, 03/2009 2009.

[Robles, 2006a] Gregorio Robles. Contributor turnover in libre software projects. In *Proceedings of the Second International Conference on Open Source Systems*, 2006.

[Robles, 2006b] Gregorio Robles. *Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, February 2006.

[Robles, 2010] Gregorio Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *MSR*, pages 171–180, 2010.

[Runeson & Höst, 2009] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, 2009. 10.1007/s10664-008-9102-8.

[Shull *et al.*, 2008] Forrest J. Shull, Jeffrey C. Carver, Sira Vegas, and Natalia Juristo. The role of replications in empirical software engineering. *Empirical Softw. Engg.*, 13:211–218, April 2008.

[Śliwerski *et al.*, 2005a] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[Sliwerski *et al.*, 2005b] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR*, 2005.

[Sommerville, 2006] Ian Sommerville. *Software engineering (8th ed.)*. Addison Wesley, 2006.

[Terceiro *et al.*, 2010] A. Terceiro, L.R. Rios, and C. Chavez. An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. In *Software Engineering (SBES), 2010 Brazilian Symposium on*, pages 21 –29, 27 2010-oct. 1 2010.

[Ukkonen, 1985] Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1-3):100–118, 1985.

[Weiss *et al.*, 2007] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[Williams & Spacco, 2008] Chadd Williams and Jaime Spacco. Szz revisited: verifying when changes induce fixes. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36, New York, NY, USA, 2008. ACM.

[Yu *et al.*, 1988] T J Yu, V Y Shen, and H E Dunsmore. An analysis of several software defect models. *Software Engineering IEEE Transactions on*, 14(9):1261–1270, 1988.

[Zhang *et al.*, 2010] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, page n/a, 2010.

[Zimmermann *et al.*, 2006] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead, Jr. Mining version archives for co-changed lines. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75, New York, NY, USA, 2006. ACM.

[Zimmermann *et al.*, 2008] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. *Predicting Bugs from History*, chapter Predicting Bugs from History, pages 69–88. Springer, February 2008.

Appendix D

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a "**Adaptation**" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b "**Collection**" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is

included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.

c "**Creative Commons Compatible License**" means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.

d "**Distribute**" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

e "**License Elements**" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

f "**Licensor**" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

g "**Original Author**" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

h "**Work**" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch

or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

i "**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

j "**Publicly Perform**" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

k "**Reproduce**" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d to Distribute and Publicly Perform Adaptations.

e For the avoidance of doubt:

i Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

iii Voluntary License Schemes. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from

any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

b You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of

the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING,
LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with

respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

f The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.