The Core of Open Source Systems

Jorge Nicolas Barrionuevo
Master of Science Thesis
Software Engineering
University of Amsterdam
Faculty of Science
Supervisors: Dr. Bas Van Vlijmen and Dr. Jurgen Vinju
Availability: Confidential

October 29, 2012

Revision History

Version 1.2: October 27, 2012 Complete version including corrections based on feedback of previous revision.

Version 1.1: October 25, 2012 Complete version including corrections based on feedback of previous revision.

Version 1.0: October 22, 2012 Complete version including corrections based on feedback of previous revision. Complete Appendices.

Version 0.2: October 19, 2012 Draft complete version.

Version 0.1: August 16, 2012 Initial version to present structure.

Contents

List of Figures

List of Tables

Acknowledgements

To begin, I want to thank Alejandra, my girlfriend, for her invaluable love, company and support during all this year.
Then, I would like to thank to those who helped me with this research project. To Bas van Vlijmen for the continuous encouragement, dedication, good listening and nice brainstorming sessions during all the thesis preparation process. To Jurgen Vinju, for sharing his knowledge and precise suggestions, that helped me find the way in difficult moments.
I would like to express my gratitude to Hans Dekkers for sharing his creativity in the conception stage of this project and for all the support, dedication and listening during the whole academic year.
Thanks to Professor Hans van Vliet (Vrije Universiteit van Amsterdam) and Chintan Amrit (University of Twente) for receiving me at their offices to discuss the project and for providing interesting ideas.
Finally, I would like to thank all those who collaborated in some extent with this research: the professors from the University of Amsterdam, who shared their knowledge, views and provided feedback, and also to Bert Lisser (Centrum Wiskunde & Informatica), Alexander Serebrenik (Eindhoven University of Technology) and Andy Zaidman (Delft University of Technology).

Dedication

To Alejandra.
To my mother Alicia, mi father Jorge and mi brother Sebastian.
To my grandmother Julia, who will always live in my memory.

Abstract

Open source systems are extensively used both in industry and academy. This widespread utilization, combined with the unique opportunities offered by these systems for outside researchers to study, both the engineering and the social aspects of the development, make this field extremely interesting. However, after several years of experimentation, there are questions that are still unanswered. It is not clear yet if those open source developers who produce the artifacts that are core to the function of the software structure are also those who contribute the most to the construction of the entire system. Understanding this correlation will increase the knowledge about the contribution patterns on which these systems are evolving, and this can help to determine how to organize roles and assign work in future projects, or suggest the possibility to replicate the pattern in other type of initiatives (i.e. proprietary software). The outcome of the present work will allow to suggest that, in the context of open source systems development, the top contributors are, in a great extent, those developers who produce the core of the system. This is a step forward to the definition of core developers that opens new possible paths for future research.

Chapter 1

Introduction

1.1 Prelude

Nowadays, the relevance of open source systems is widely accepted both in industrial and academic circles and in the last 20 years several research works have been conducted to study their properties. One particular characteristic related to these systems is the core-periphery arrangement, that can be studied from two different perspectives: the technical-architectural and the social (aspects involving the people that is part of the development process, their activities and interactions).
On the one hand, the technical-architectural core of the software is usually made of the artifacts that are critically important for the system. When these entities exhibit high levels of connection or dependency with other components, that is, when the artifact that is being studied is both using and being used by other software entities. On the other hand, the technical-architectural periphery of the system is defined as a composition of artifacts that are less strongly connected to other artifacts.
Considering the contribution of the developers that produce the technical core of the open source systems, it could be claimed that these core artifacts are generally created and maintained by the developers that are key to the project in terms of participation, commitment, decision power and access rights. In the same line of reasoning, it could be argued that the artifacts that constitute the technical periphery of the system may tend to be created or maintained by developers that contribute less to the project, have less decision power and access rights to the critical artifacts of the system. These are called peripheral developers. However, the presence of this pattern of contribution will depend on other factors as the governance style of the project (*a priori* definition of when, who, which roles, and in what extent they are allowed to access to the core), the level of development centralization (extent on which one developer do disproportionately more than the rest), the particular necessity in certain stages to count with core developer participation in the periphery (i.e. near release dates) and the software architecture (how well the system allow to developers with different levels of knowledge and experience to work simultaneously in different parts of it).

The main goal of the present study is to determine if a correlation between the participation of top developers (those who contribute more to the project) and those developers who build and evolve the technical core can actually be distinguished. This would favor the understanding of the contribution patterns of open source systems and, consequently, to help determine how future projects of this or other types may be organized.

1.2 Motivation and Relevancy

The study of core-periphery properties of open source systems is relevant in several aspects.

First, understanding contributions in open source projects is useful to facilitate the comprehension of the characteristics of the processes that are implemented to build these systems. Therefore, it can help to increase knowledge relevant to improve coordination, work assignment and, consequently, to help take decisions aimed to avoid the failure or to promote the success of projects. This is important in a context where outsiders, with no prior knowledge of the project, need to gather information from the software system and processes in order to make decisions.

Second, it allows to increase the understanding of the coupling characteristics of the systems. These properties are directly related to the modularity of the software [18] and hence to the propagation of changes across them, which in turn, have managerial implications, as maintenance costs, work assignment and resource allocation (associated with the independence or interdependence of tasks and communication among members and teams). The more coupled are the artifacts of a system, the more difficult it would be to understand, change, and correct this system [5]. Thus, understanding core-periphery features of software systems favor the comprehension of the modularity and the structural complexity they present.

When studying open source projects, several publications determine the most important developers in terms of amount of activity [20, 25, 9, 22]. One may question if it is a good definition of a key (core) developer. Intuitively they can be presented as the individuals who work on the most important parts of the system, those parts that require the most experienced and knowledgeable contributors.

While working in the present study and reasoning about the definitions of the different roles of developers in relation to the core of the systems (core or key and top contributors) it was not possible to find a research that focused on validating the assumption that, the core of the systems, is developed by the most important contributors of the project. Just two academic publications [1, 22] that define core developers in relation to their affiliation to the technical core were found. Amrit *et al.* [1] propose that developers may be defined as core or peripheral in terms of the part of the system they work on (core or periphery) but it is suggested further research to validate this definition. The only authors who apply and try to validate the assumption are Oliva *et al.* [22] where core developers (treated as “ key”) are those who work on the core, and, it was found that this group is made by the same who commit or work the most. The main problem in this case is that the study was conducted on a single open source system, reducing the extent of generality of the conclusion. In summary, the definition was not defensible (could no be used in other studies).

The relevancy of the present research work relies on identifying a correlation between the total contribution realized in an open source project and the developers who produce the technical core of these systems. In other words, it favors the understanding of the relations existing between the definitions of top contributor and core developer.

1.3 Research Question and Hypothesis

Following the line of reasoning presented in the previous section, the research question is introduced:

RQ: *Is the technical core of open source systems being developed by the top contributors?*

In turn, this question led to the following hypothesis:

H: *In the context of open source systems, the top contributors tend to be those who produce the core.*

This expectation is deducted from the definition of core developer. On the one hand, it is reasonable that those who produce in a greater extent and are more committed to the project, also have the capacity, knowledge and access rights to the areas of the software structures that are critical for the system as a whole. On the other hand, it should be regarded that according to how the projects are organized in terms of core access rules, role advancement policies (in the community), style of management or level of development centralization, this may vary.

In summary, some publications assume the core to be the most tightly coupled part of the system and the core developers as those who most contribute but: Do they actually correlate? From this, in Figure 1.1 the conceptual model of this study is presented.

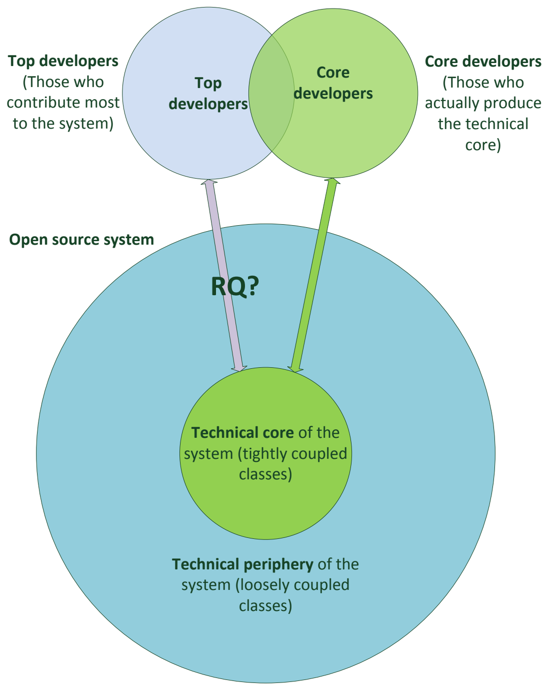
Top developers
(Those who contribute most to the system)
Top developers
Core developers
Core developers
(Those who actually produce the technical core)
Open source system
RQ?
Technical core of the system (tightly coupled classes)
Technical periphery of the system (loosely coupled classes)

Figure 1.1: Conceptual model of the research study.

In this context, it is important to mention that the model of Figure 1.1 is just an abstraction of the structure the systems may present, as they can be actually composed by multiple artifacts (or cores) that have high coupling properties. This abstraction is aimed to favor the analysis and understanding of the contribution patterns.

1.4 Research Method

In the next section, a summary of the research method is presented. The method will be repeated for each case study.

1. Select an open source system to be studied.
2. Identify software revisions to divide the study in time frames between them.
3. For each studied sub-period of the project:
    (a) Measure the coupling levels of each class of the technical structure.
    (b) Aggregate the coupling levels from class to source file (for each class that is part of a Java file, calculate the sum of each metric value independently).
    (c) Define a coupling threshold to group the Java files (the ratio at which each metric value will be considered high or low).
    (d) Determine the presence of a static-coupling core-periphery structure in the software.

(e) Define the core of the system by creating a set of the Java files that, according to the defined threshold, present high coupling in all the metric values.

(f) Measure contribution of each developer (as author) to the whole technical structure studied. Calculate the total contribution of the period.

(g) Measure the contributions related to the technical core of the system. Those developers who produced the core will be defined as core developers.

(h) Verify the extent (correlation) to which the whole technical structure (the system) was developed by core developers.

4. Measure other variables related to the technical structure that will be appropriate to understand the results of the period.

5. Measure other variables related to the contribution in the period that will be convenient, to analyse the results.

According to the presented research method, on one hand, if the developers that are identified as core (3.g) are found to be producing a high proportion of the total contribution realized in each period (3.f), this will be an indication of a strong relation between core and top developer groups (core developers are contributing the most to the whole project). On the other hand, if the amount of total participation (3.f) produced by those who are evolving the core (3.g) is found to be low, this can be considers as a sign of weak relation (core developers are not producing the most of the total contribution in the project).

Chapter 2

Theory

2.1 Core and Periphery in Open Source Systems

From a network theory perspective, a core periphery structure can be defined as a network where a reduced number of central entities gather a disproportionate amount of connections, while most other entities maintain few relationships [12]. As this type of structure is common in social networks [12] it is expected that a piece of work (i.e. open source software system) that is the product of social interaction, will preserve the properties of (or correlate in some way to) the structure under it was conceived (i.e the Conways' law [8]).

Borgatti and Everett [4], also based on a social network approach, and starting from the accepted assumption that the core is dense, cohesive and the periphery is sparse and unconnected, intent to formalize this and other intuitive definitions of core and periphery into discrete and continuous models. One definition assumes that in a network of nodes, all of them belong in a greater or lesser extent to the network, some entities may be better connected than others but it is not possible to make a partition where one group is cohesive and the other is not. The other intuitive definition is the notion of two class partition where one group is the core and the other the periphery.

The present study, though it will not have a social network approach, will base on the two class partition concept, so one group of artifacts will be considered core and the rest non-core or periphery. This study has been scoped in this way because the focus will be on the contribution of those software developers that produce the artifacts that have core properties and it would not help to apply a continuous analysis.

Mac Cormack *et al.* [18] made a study over 19 complex and successful applications (in terms of size and number of end users respectively) and found that in most of the cases a technical core-periphery structure was present. The research covered systems with different languages and was conducted at module aggregation level. It was found that the amount of modules in the core may vary among systems (even performing the same function) and that the size of the core across the evolution of the system may be stable or may grow in proportion to the rest of the software structure. The publication defines core components as those that are tightly coupled to other components (high fan-in and fan-out visibility) and, in contrast, peripheral components as those that are loosely coupled to other components (low fan-in and fan-out visibility). Coupling is measured by creating a call graph of the system and by counting direct and indirect calls in both directions but no other consideration regarding the characteristics of the studied language (as inheritance or field access) are taken into account.

In the present research, it was decided to limit the scope of the study of the technical structures to the characteristics of a defined language, object oriented in this case (Java). It would be difficult to form judgement from results that are product of measuring properties of the software artifacts written in languages with different characteristics. From this perspective and in order to define the artifacts that are core to the system function (a system with particular properties inherent to the utilized language), it is important to:

1. Consider an adequate aggregation level.
2. Understand the specific coupling mechanisms between artifacts.
3. Utilize the appropriate measures to quantify the connections.

In the present work, unlike Mac Cormack *et al.* [18] and in order to characterize core artifacts, it was decided to work at class level (instead of modules) and to measure other properties that reflect the conceptual definition of the core, in terms of object oriented coupling (as class and method relations instead of only fan in and fan out dependencies).

Oliva *et al.* [22] define the technical core of the studied system using dependency call graphs. Then the key developers are defined in terms of their volume of contribution to the technical core and their social participation (activity in the mailing list). It was found that only 25% of the developers may be considered as key and that there is no difference between key developer and top contributor set (the ones who most contribute are same who access to the technical core of the system). In the present research work, this correlation between those who most participate and those who work in the core will be focused and validated in more systems (Oliva *et al.* [22] found and studied the relation in just one case).

Amrit and van Hillegersberg [1] studied the socio-technical movements in open source projects. It was found that when developers contributing to the periphery move towards the core across the evolution of the system, it is beneficial for the project, in contraposition to shifts away from the core that are not good for it. The paper studies which developer is working in each part of the technical core-periphery at any given point in time and relate the shifts to the interest that developers have in the project. On one hand, the author defines the technical core as the more dependant part of the code in terms of class and function dependencies: a modification on a core module will affect more core modules than when working on the periphery. On the other hand, developers are defined as core or peripheral in terms of the technical parts of the system they are related to. In this work, several Java open source projects with diverse characteristics (in terms of domain, size and community) were studied. In the present research, a similar approach will be utilized, first defining the technical core (though using object oriented metrics) and then analyzing the participation of developers in this structure.

2.2 Coupling in Object Oriented Systems

Coupling is a concept that was introduced in the context of structured development, as “the measure of the strength of association established by a connection from one module to another” [23]. If the modules are tightly coupled with other modules (more inter-related) the system will be more complex and consequently, the resulting software will be more difficult to understand and maintain [5]. However, coupling is not an exclusive characteristic of modules in structured development but it also applies to object oriented systems (and to their artifacts). In the latter case, coupling is a more complex property due to the diversity of mechanisms that can constitute it [5].

The concept of coupling is directly related to another important architectural definition, the modular decomposition, that characterizes how a design is separated into modules. A system is modular, to an extent, if the modules that constitute it are strongly interdependent within themselves and weakly dependent to other modules [18]. This is what is called loose coupling. This concept is important because, in a modular design, a modification to a module is expected to have less impact in others [18] and this will favor the maintenance process and may add value to software designs (by creating options to improve the system by substituting or experimenting on individual modules [2]).

2.2.1 Object Oriented Metrics

In order to measure the levels of coupling of object oriented systems, several metrics have been defined (i.e. Chidamber and Kemerer [7] or Martin [19]). These metrics can be classified into two groups: static and dynamic. Despite some publications consider dynamic metrics as more precise [26], in the context of the present study it would be not possible to implement this type of metrics. Dynamic coupling measurements require the study of the execution environment for each software system analyzed and the selection of a set of relevant scenarios to be measured. This approach has two disadvantages: it is very expensive in terms of time for research and there will be areas of the system that cannot be covered (due to the limitation of the scope to a set of scenarios or, for example, in the case of “dead” code, that is still relevant in terms of contribution). These were the reasons why it was decided to conduct a static study to understand the coupling characteristics of the software systems.

An important feature of coupling is the direction of the dependency. In this sense there are two types of coupling: import and export. The first refers to the use of services provided by others, and the second refers to providing services to others. Zaidman *et al.* [26] propose a method to identify “key classes”, that are those important as a starting point to help software engineers (who are new to a project) to understand the inner workings of the software architecture. These classes are defined as those that have a controlling role (give orders to other classes), and this is represented in terms of strength of import coupling. The other approach is proposed by Mac Cormack *et al.*[18], where coupling is defined in terms of direction as fan-in (export) or fan-out (import) and in terms of strength. Under this denomination, four types of categories of components are defined:

1. Core (high fan-in and high fan-out).
2. Shared (high fan-in and low fan-out).
3. Control (low fan-in and high fan-out).
4. Peripheral (low fan-in and low fan-out).

In the present work, the "coreness" of the artifacts of the system will be defined in terms of high levels of both import and export coupling as Mac Cormack *et al.* did [18] because the focus will be on the importance of the artifacts to the system function and not on the cognitive level (to facilitate program comprehension for new software engineers).

Chidamber and Kemerer [7] produced a classical work on the theory behind object oriented metrics. In this publication, the author defines Coupling Between Objects (CBO) as a metric that measures the number of other classes to which a class is coupled to, including inheritance ("a method m of a class c uses a method or attribute of an ancestor class of c" [7] or in other words "because an inherited method is considered a method of another class."[5]). He also defines Response For a Class metric (RFC), that count the response set of a class, that is, the set of methods that can potentially be executed in response to a message received by an object of the measured class.

Briand *et al.* [5] studied existing object oriented coupling frameworks. This publication standardizes the related terminology and analyses motivation, empirical hypotheses and application of several coupling metrics. The authors propose a unified framework based on the issues discovered. This paper will be used as a source to support the applicability of the existing metrics (and the necessity to extend them with new features or the need to utilize other measures) to the context of technical core-periphery definition. In this study, CBO and RFC metrics are mapped to the domain of class measures. In the context of the present work, as CBO is regarded as a measure of both import and export coupling and as RFC is utilized to gauge export coupling [5], these metrics will be regarded as suitable to evaluate core properties.

Martin [19] presents object oriented design concepts as category, responsibility, independence and stability (of a category). A class category is a group of cohesive classes that will tend to be modified together (that is, if one class has to change, all the classes in the category are also open to change). These classes are reused together because they are strongly interdependent, and they also share a common function or goal to achieve [19]. The author presents object oriented metrics useful to measure the quality of the design in terms of responsibility, independence and stability. These metrics are Afferent Coupling (Ca) and Efferent Couplings (Ce). On one hand, Ca measures the number of classes outside the category (on which the measured class inhabits) that depend upon classes within its own category. On the other hand, Ce measures the number of classes inside its own category that depend upon classes outside this category.

In relation to the purpose of the research, Ca will be utilized to measure how many classes are using a measured class, and Ce purpose is to compute the number of classes that the measured class is depending on. Thus, these measures, together, will reflect the number of bidirectional dependencies that define the core (in terms of inheritance, field accesses, method calls, arguments, exceptions and return types).

Finally, and with the purpose of excluding from the focus of the measurements exceptions and return types (to avoid deviations that could be caused by them), it was decided to include two extra class coupling metrics. In subsequent sections these will be denominated as inbound and outbound dependencies. On the one hand, inbound dependency metrics refer to the classes of the same and other packages that depend on (use or import) the measured class. On the other hand outbound dependencies refer to the count of other classes that the evaluated class is depending on (using or importing), both in the same and in other packages. This metric reflect class coupling based on the count of dependencies to other classes, their fields and methods.

In the present research work the introduced metrics will serve to implement the coupling concepts required to define the technical core of the systems. These object oriented metrics are CBO, RFC, Ca, Ce, inbound and outbound dependencies.

2.2.2 Clustering the Core

Once the metrics that reflect the properties of the core are measured, and to facilitate the study of the contributions of developers to them, it is necessary to generate an abstraction of the technical core. In order to do this it is necessary to:

1. Change the aggregation level to source (Java) files.
2. Cluster the source (Java) files.

On the one hand, the first item is required to match the aggregation level found in the software repositories. On the other hand, the clustering or grouping of the files is required to enable the focus of the contribution study in a single structure defined as a core (though there could be actually multiple ones) and in those who develop it.

In order to cluster the files to represent the core-periphery dichotomy, it is required to define a metric threshold value and to group the files of the system based on the coupling level of all the metrics. Therefore, all the files that present values that are above the threshold, for each of its metrics, will be considered part of the core.

As every mentioned metric have its own motivation and empirical hypotheses, this threshold value must be calculated for each of them. That is why the threshold will be defined as a ratio of the total range and then translated to absolute values for each metric. In this way the conceptual independency of metrics is preserved.

2.3 Contributions in Open Source Systems

The onion model has been accepted as a representation of the distribution of contributions in open source projects [20, 10]. In this model a group of developers that do most of the work and are responsible for important decisions are considered core, while, other group that contribute less frequently and use to have less decision power is defined as the periphery [25]. In this sense, the model is directly related to the concept of development centralization, namely, the extent to which projects will have code written by a small group of individuals, which is an important aspect of open source project development [10].

Other definitions of core developer can be found in different studies. Amrit and van Hillegersberg [1], for example, present an integrative socio-technical approach. It bases the definition of architectural core and periphery on call graph (function calls) and class dependencies. Core and peripheral developers are defined in terms of which part of the software they work on. If a movement of developers that work on the core towards the periphery is not followed by a movement of developers that work on the periphery towards the core it is interpreted as negative for the project (key developers are loosing interest). This "shift" of the contributors working in the core is considered an indicator of the health of the project. The study proposes that further research should be done in order to asses the quality of the assumed definition of core-peripheral developers in terms of technical affiliation. Following this line of work, the present study will try to find the correlation between the top contributors and core parts of the systems to help assess the mentioned definition.

Terceiro *et al.* [25] define core developers as the most active, the ones who perform most of the work and are responsible for the most important decisions of the project. Peripheral developers contribute less frequently and have less decision power. The present work will use the definition of core developer in terms of activity and amount of work but in this case, and from now on, will be denominated as "top contributor". This notion will be compared with the definition of core developer in terms of affiliation to the core codebase (those authors who produce and evolve the core).

Crowston *et al.* [9] present 3 methods to define core developers of FLOSS projects and compare them: the claimed list in the documentation, the frequency of contributions to the bug tracker system and a measure related to the social network activity. The three techniques suggest the core is a small fraction of the total number of contribution (5%) but different individuals are identified as core developers by each method so it is suggested that the measures should be refined. It was found that developer list is not a good measure to define core contributors (from bug tracking systems) and that the skew of contribution in the communication domain is higher than in the code domain.

A socio-technical approach is proposed by Oliva *et al.* [22] to identify key developers of an open source project. The method is based on the analysis of different time frames in the history of the project and utilizes call graphs to define the core artifacts and commit "coreness" to define key developers (amount of contribution to the core). Then the study focuses on social aspects as top contributors, communication network, coordination requirements network and congruence between them. What is particularly relevant for the present work in this study is that the identified group of key developers is exactly the same as the set of top developers: those who work on the technical core are the same who made the most part of commits. In the present study, the definition of key developer (commit contributions to the core) will be compared with the definition of top developer (commit contribution in a period to the whole system) to understand if these are correlated.

Finally, Huang *et al.* [13] defined the core developers in relation to the common directories they work on (in an affiliation graph) but the correlation to the technical core is not studied. From this work it is not possible to state that the core of the system (as tightly coupled artifacts) is produced by the top developers of the project.

In the present work, the main goal is to answer if those who contribute to the areas of the systems that present core properties (high levels of coupling with other artifacts) can also be considered core as it is defined by Terceiro *et al.* [25] and Crowston *et al.* [9] for the case of amount of contributions. In other words, the idea is to determine what is the impact (in terms of contribution) that the developers who produce the core have in the rest of the system.

Summarizing, in previous publications core developers were defined as those who produce or modify the core of the systems [1, 22]. It was suggested the necessity to validate this definition [1] and it was found that, at the moment, this had not been made with a reasonable extent of generality [22]. The present work is aimed to increase the understanding of how core and top developer groups relate by producing a research focused in this particular issue and by studying a larger sample of open source systems.

Chapter 3

Methods and Tools

3.1 Outline

Due to the characteristics associated with the evolution of software systems and the projects under which those are created, it would be difficult to draw precise conclusions and to understand contribution patterns related to parts of these systems, just by computing total participation, or by assuming that the software is made of a set of permanent artifacts. These projects may have been developed during long periods (i.e. years) and, in each of them, there could be found developers arriving and leaving, artifacts being created and deleted, roles being changed (i.e. promotions), differences in productivity levels (contribution), variation in the access rights to certain areas of the code etc.

These changing characteristics are the reason why the present research has a software evolution approach that will allow to produce conclusions based on the understanding of how the projects mature in relation to the mentioned aspects. In this context, it means that both the study of the technical properties of the core and the study of contributions will be done in intervals between software revisions.

In order to realize the experimental stage of the study, it was necessary to design and develop a software application to automate the extraction of facts concerning both the software systems and the contributions realized by developers.

The application was designed with two independent modules: the core-periphery and the MSR modules. In the case of the first of them, it was divided into two submodules, one for the implementation of the object oriented metrics and the other for computing dependencies (see Figure 3.1). In the case of the latter, it is aimed to process the data from the version control systems, where the sampled projects reside.

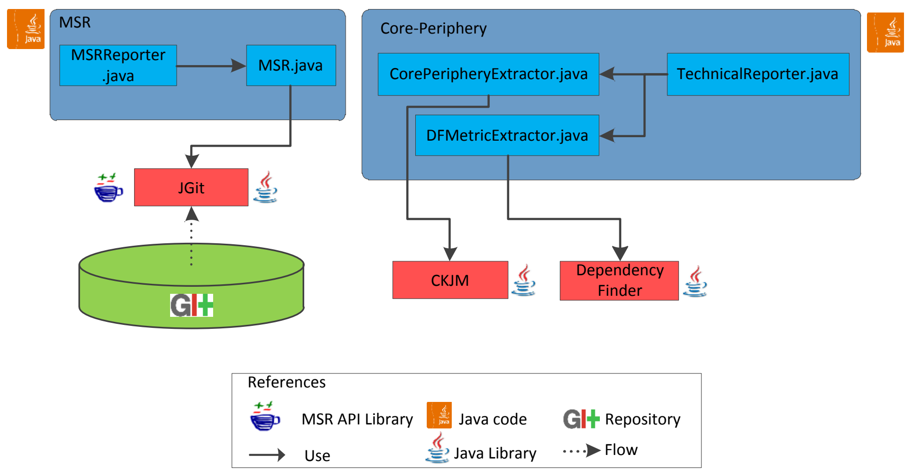

Figure 3.1: Application design for the experiment.

In the following sections, the implementation decisions of the core detection and MSR tools will be explained.

3.2 Finding the Technical Core

It was decided to conduct the experiment at class aggregation level because some of the well known coupling metrics for object oriented systems [7, 19] made focus in classes. In order to integrate the outcome of both the technical core extraction and the MSR tools, it was necessary to add an extra process to aggregate from class to source file level (Java files). In the present experiment it was decided to address this aggregation by calculating the sum of the values of each metric (independently).

Although this stage is required, it could become a threat to validity in the cases where there are classes with a disproportionate number of inner classes that present low coupling. Despite this, the sum approach was preferred to aggregating with average measures because, in the case that an outlier class (in terms of coupling) is found in a source file with a great amount of other inner classes (that present very low coupling measures), the calculation of the average will make loose track of this core class.

Given the defined requirement to utilize the mentioned static metrics, and due to the outcome variations mentioned in previous work [17], it was decided to not implement the metric specifications from the beginning but to reuse other implementations from two existing open source projects already employed in academic research [24, 15, 16, 1]. These projects are CKJM[1] and Dependency Finder[2].

The introduced metric tools require the bytecode files (.class) as the input of the process. Focus will be kept on the classes that constitute the central part of the software binary distribution and those classes that are foreign to the project (libraries) will be discarded.

[1]http://www.spinellis.gr/sw/ckjm/
[2]http://depfind.sourceforge.net/

3.3 Computing the Core

The core of the systems is determined by measuring the coupling levels of all the metrics introduced in Chapter 2.3. For all the classes that are part of a source file, the aggregated values are calculated (addition of each of them by metric type). The Java files that present values above the threshold are considered tightly coupled and grouped into sets per metric. Finally, the core is defined as their intersection, in other words, the files that present measures over the threshold for all the metrics (see description of metrics in Appendix A).

For example, in the case of Jenkins project it was found that the following files are part of the core:

jenkins/model/Jenkins.java,360,1023,386,360,28,117
hudson/FilePath.java,262,708,246,262,101,163
hudson/model/Queue.java,150,446,132,150,26,48
hudson/util/ProcessTree.java,128,408,93,128,28,64
hudson/model/UpdateCenter.java,158,432,90,158,40,73

While the following is a sample of files that are not part of the core:

hudson/model/FileParameterDefinition.java,12,24,0,12,0,2
hudson/model/WorkspaceListener.java,7,7,2,7,0,2
hudson/init/impl/InitialUserContent.java,3,9,0,3,0,1
hudson/cli/GetJobCommand.java,7,9,0,7,0,2
hudson/search/CollectionSearchIndex.java,4,15,4,4,4,2

In the first case, the values of the metrics related with the core are high in relation with the second example (non core).

3.4 Mining Software Repositories

In order to produce the results for this type of research it is important to avoid some methodological issues that emerge from the use of self reported data and statistics offered by the project members or the forges that host them (like SourceForge or GitHub)[11]. This data may be problematic and biased and that is why in the present study it was decided to extract and analyze it directly from the version control systems where the sample of open source projects resides. This task is time consuming but it is worth the effort to overcome these potential threats to validity and improve the objectivity of the measures.

The fact that open source projects may be available in different version control systems and the need to produce a software application to process the required data in a limited amount of time, led to the necessity to select a version control system that satisfies the requirements of the research. In this sense the most important aspect to evaluate is the accuracy of the version control system to attribute contributions to software developers. The log of the system must reflect the diversity of collaborations and this is not possible if a centralized version control system is used because, in this case, the logging of the activities of contributors outside the group of commit right is not possible.

In Git, the authors may contribute a modification in two different ways.

First, they can clone the repository, make modifications to their working copy and ask the developers with commit right to the main repository to pull the changes to it. Second, they can create a patch with a commit (or a sequence of them) and send it to the developers (by email for example) with access to the central repository so they can apply the patch to it. In both cases the authorship information is always available. That is, the author information travels through repositories and is preserved. This is possible in Git because there are two fields available to reflect contribution, *committer* and *author* [3]. Bird *et al.* [3] the number of authors of a project that started using SVN and then migrated to Git were studied. It was found that the amount of authors per period increased after the migration to Git and this may be attributable to the better authorship tracking facilities offered by this version control system.

Another important advantage of Git is the way it treats data. Git stores data as a snapshot of the files in the commit and then references it (the files that are not modified are not stored again). Other version control systems store the data as a list of file based changes [6]. Other advantage of Git is that the possibility to work on local repositories makes almost all operations local, and this reduces the network latency overhead [6]. These features are the main reasons for choosing Git in the present research work.

In order to create a flexible tool for mining the software repositories, it was decided to build it on top of the Java Git open source API [3], also, already used in other academic works [21].

3.5 Computing Contributions

The method implemented to measure contribution of a developer consists in counting the number of times (where the developer performs as author) that a Java file is modified. The total contribution of a period is defined as the sum of all the instances of work regarding all the developed or maintained files present in the sampled structure. In the same line, the contributions to the core are those instances of work that are made in the files that constitute the technical core.

3.6 Sampling Strategy

In order to conduct this research, the selection of the open source projects to study had to satisfy certain characteristics, to assure that the contribution and properties of the technical core could be measured. The systems needed to be large enough in order to offer a degree of complexity, count with at least 4 revisions to exploit the benefits of the software evolution approach (study of different time frames), having evolved throughout a period of more than 3 years of development (also to explore evolutionary aspects) and having at least 15 developers to improve the understanding of contribution patterns (it would be more difficult to draw conclusions if the studied project has, for example, one or two developers).

Also, the systems must be diverse in size (lines of code), amount of developers and application domain to improve the extent of generality of the conclusions

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

(no particular focus in a type of system). The systems that were selected as data samples for the present study are presented in Table 3.1

System	Developers	Time frame (days)	Revisions	LOC
Jenkins	285	4,473	11	40,667
Rascal	27	1,372	5	81,201
Clojure	98	2,210	4	2,640
Oscar	52	3,225	9	105,431
Solr	17	1,890	5	42,796
Voldemort	61	1334	10	36,892

Table 3.1: Open source systems sampled.

Regarding the technical structure, the dataset studied presents the characteristics introduced in Table 3.2

System	Packages	Files	Classes	Inner Classes
Jenkins	56	815	1,804	891
Rascal	94	963	2,134	1292
Clojure	21	164	759	549
Oscar	173	1,405	1,518	103
Solr	41	671	1,191	414
Voldemort	69	385	672	284

Table 3.2: Technical-structural properties of the sample.

Chapter 4

Results

4.1 Jenkins

4.1.1 Case Presentation

Jenkins [1] (formerly Hudson Labs, until early 2011) is a continuous integration server. It is aimed to support developers in the building and testing processes of software applications. Jenkins posses a plugin architecture and has many plugins available. The project has a total of 285 developers that participated throughout its life cycle (almost 6 years). It has a robust community behind and is used by several companies and open source initiatives [2].

4.1.2 Results

As Figure A.1 shows, the core of Jenkins represent a small part from total in terms of Java files. The core developers are a fraction from the total (both groups maintain a quite direct proportional relation, see Figure 4.3) and they produce the most of the total contributions in all the studied time frames of the project (Figures 4.1 and 4.2).

[1]http://jenkins-ci.org/

[2]https://wiki.jenkins-ci.org/pages/viewpage.action?pageId=58001258

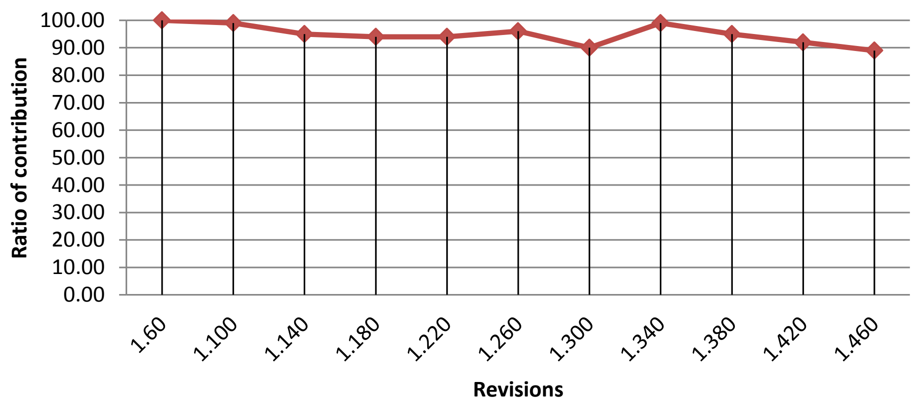

Figure 4.1: Ratio of contribution of core developers of Jenkins (from total).

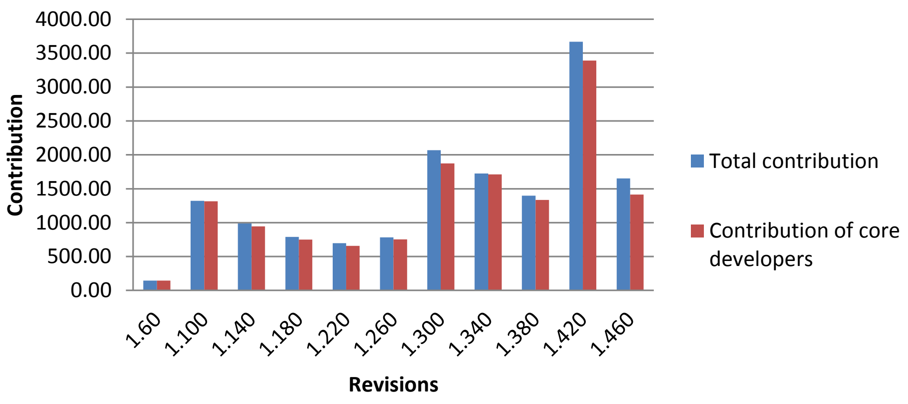

Figure 4.2: Total and core developer contribution of Jenkins.

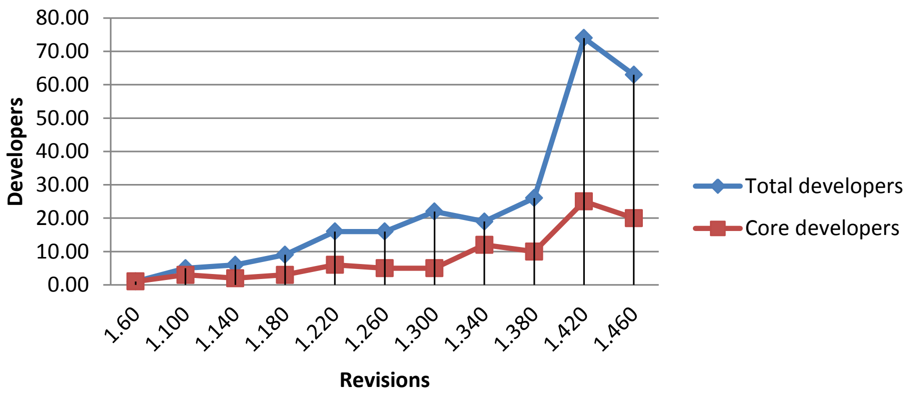

Figure 4.3: Total and core developers of Jenkins.

In terms of contributions to the core, it can be observed that in this project there is one developer that does most of the work. In Figures 4.4 and 4.5 it can be observed that this author produced the greatest extent of the core and performed the most of the total contribution in a particular time frame. This scenario is repeated in all the studied periods of the project.

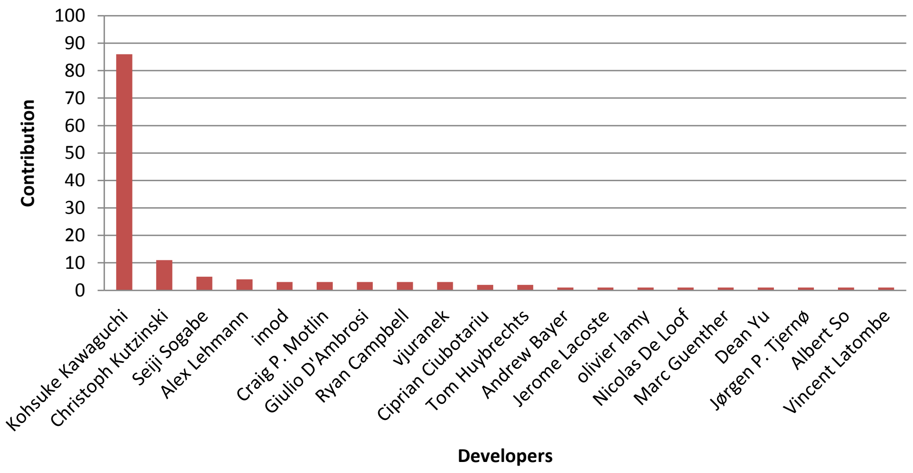

Figure 4.4: Contributions to the core of Jenkins (in time frame preceding revision 1.460).

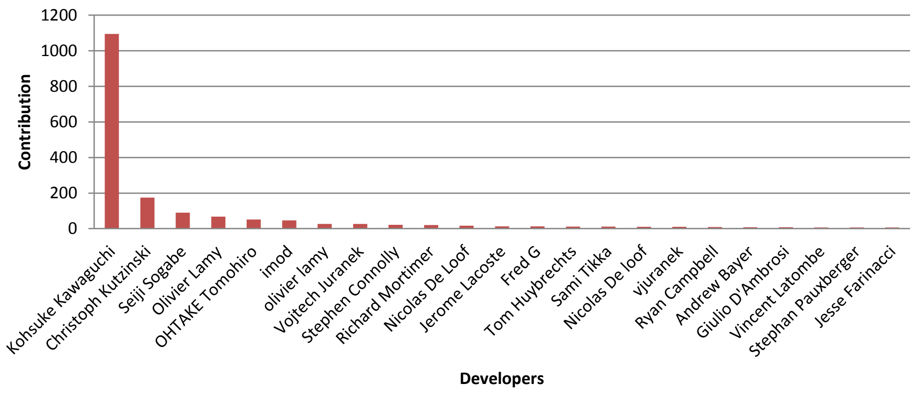

Figure 4.5: Total contributions to Jenkins (time frame preceding revision 1.460).

4.1.3 Analysis

This project is the biggest in relation with the number of contributors (see Table 3.1) and exhibits a pattern were most of the work is done by one developer (Figure 4.5). This arrangement of contribution, with one person doing disproportionately more than the rest, must be regarded, because it may mean that this developer is working everywhere (he may be not specially affiliated to the core). In this sense, it would be interesting to see if the correlation of

Figure 4.1 could be found also in systems where the contribution is more evenly distributed.

This project seems to be doing well in reference to the community, in the sense of generating interest and attracting new developers. As can be noticed in Figure 4.3 the number of developers is growing steadily. Just as the amount of contribution as seen in Figure 4.2 is not showing the same growth tendency and ratio, it is suggested that the addition of more developers does not imply an immediate increase in productivity. On the contrary (and in accordance with the Brook's law [14]) as the total contribution is fluctuating, it may mean that during some periods, top (core) contributors may be assuming other responsibilities related to the control, communication, education or coordination that the new members require.

As the productivity is oscillating, but presenting a tendency to increase, it could also mean that new developers are eventually overcoming the learning stage (becoming productive) and as they know how and what to do, the communication overhead is reduced and the top (core) contributors can be released to an extent where they can direct again their efforts to development tasks.

It also can be observed that the number of developers and authors is the same throughout most of the studied period except in the last two revisions (1.420 and 1.460) where the project changed its name from Hudson to Jenkins (see Figure B.1). This could mean two things. It could be either that before this point in the life of the project it was not part of the process to register authors in the commit or that the project was migrated from other version control system to Git, so in previous commits it was impossible to distinguish contribution type (committer or author).

In reference to the core, it can be seen in Appendix A that during the evolution of the system the coupling levels are growing. What is interesting in this case is that this tendency is followed by a growth in the number of developers that work in the core of the system. On one hand, this may be an indication that the core is becoming more coupled to other parts due to the growth of the system as a whole in terms of artifacts (see Figure A.1) and consequently deriving in an expected raise in the number of connections. On the other hand the increase of coupling may mean a lack of modularity (though to conclude this it would be necessary to measure also the cohesion of the classes) and therefore a bad response to the increase in the number of developers, in terms of coordination and division of labor.

4.2 Rascal

4.2.1 Case Presentation

Rascal is a domain specific language to perform software analysis and manipulation (meta-programming)[3]. The project, that features 27 total developers (throughout its life cycle), has been in development for more than 5 years [4] and is realized mainly in the Centrum Wiskunde & Informatica (CWI, the national research institute for mathematics and computer science in The Netherlands).

[3]http://www.rascal-mpl.org/
[4]https://github.com/cwi-swat/rascal

4.2.2 Results

In terms of contribution it can be appreciated that, although the correlation is still high (between 95% and 48%) the rate of total participation of core developers is steadily dropping (see Figures 4.6 and 4.7).

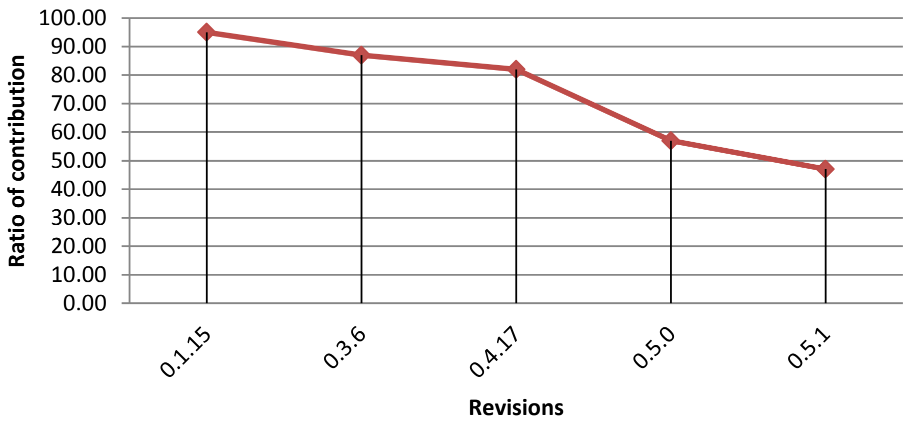

Figure 4.6: Ratio of contribution of core developers of Rascal (from total).

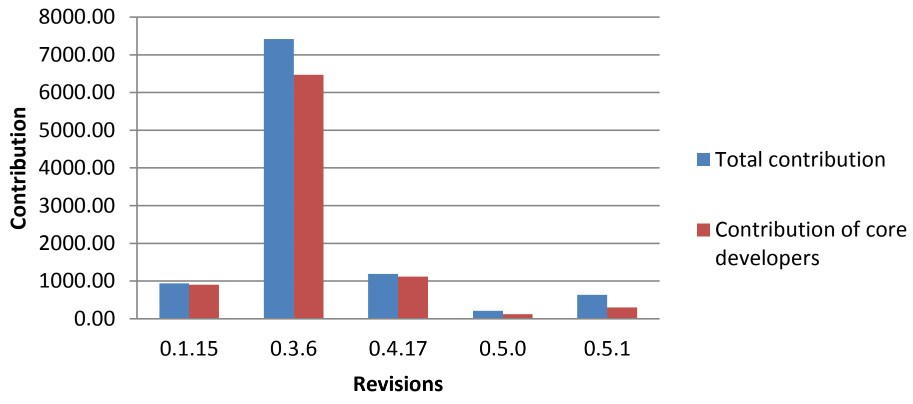

Figure 4.7: Total and core developer contribution of Rascal.

It can be seen that in the first time frame studied the core developers correlate very good with top contributors (see Figures 4.8 and 4.9), with four out of five core authors precisely matching the group of those who participate the most in the project.

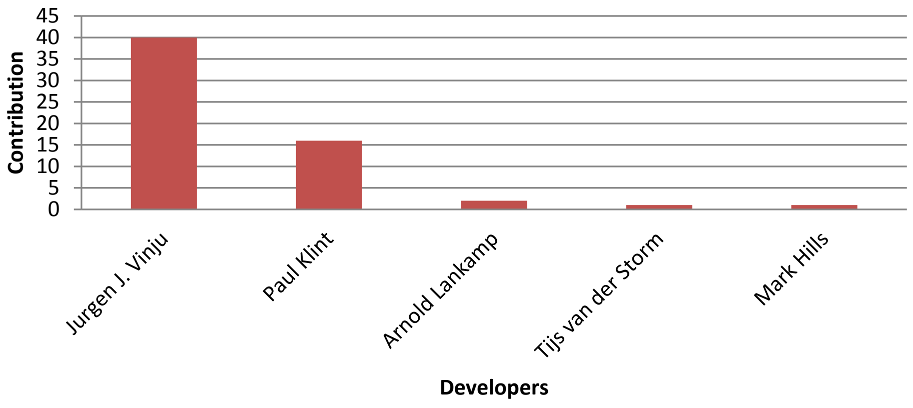

Figure 4.8: Contributions to the core of Rascal (in time frame preceding revision 0.1.15).

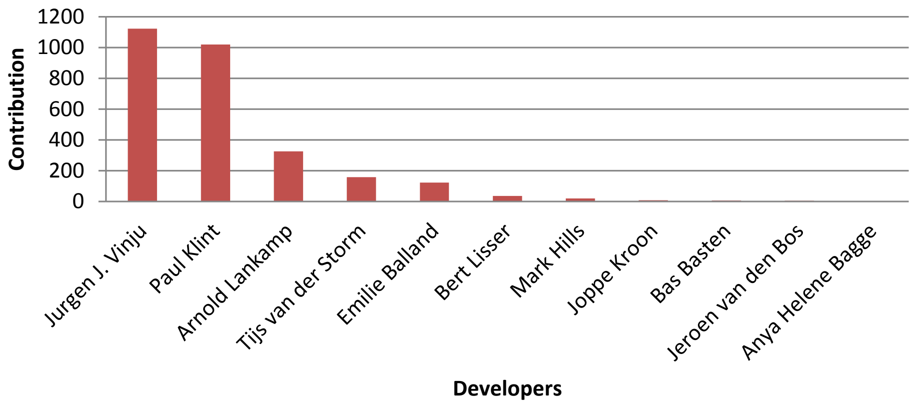

Figure 4.9: Total contributions to Rascal (time frame preceding revision 0.1.15).

In the last studied period, it can be appreciated that the correlation is present in a lower extent because a developer, that is not in the core, is contributing significantly to other parts of the system (see Figures 4.10 and 4.11), in this case, essentially libraries (to org/rascalmpl/library/ package).

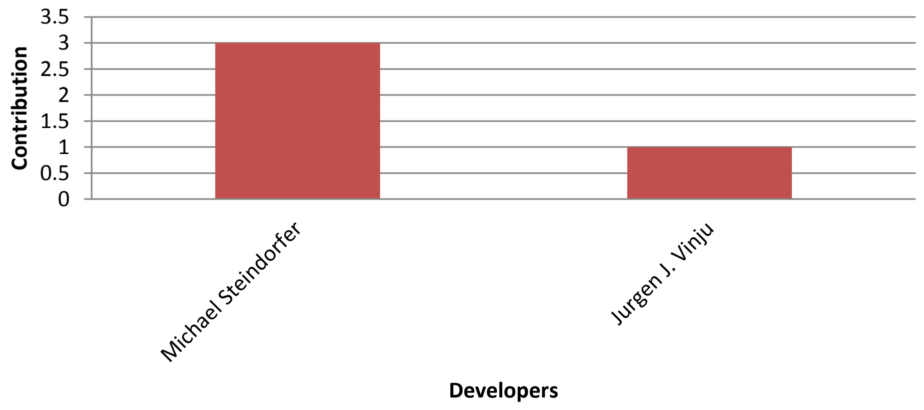

Figure 4.10: Contributions to the core of Rascal (in time frame preceding revision 0.5.1).

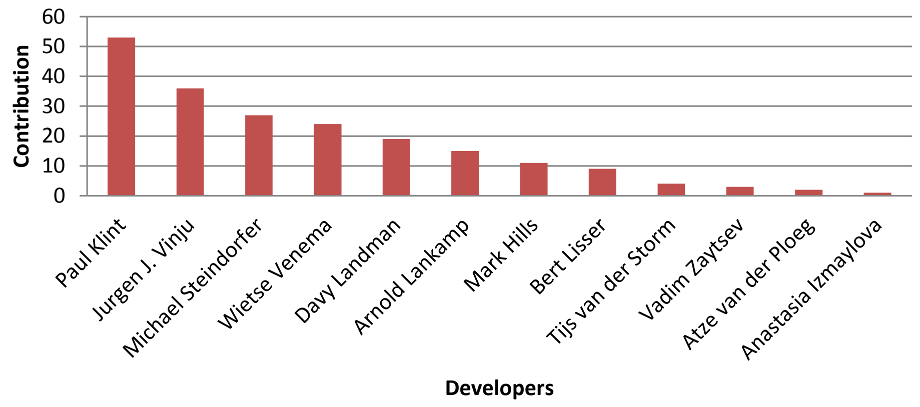

Figure 4.11: Total contributions to Rascal (time frame preceding revision 0.5.1).

In this sampled system, it was found a practically direct relation between core and top contributors (see Figure 4.12) with a decrease in the amount of developers that participate in the core in the first two periods.

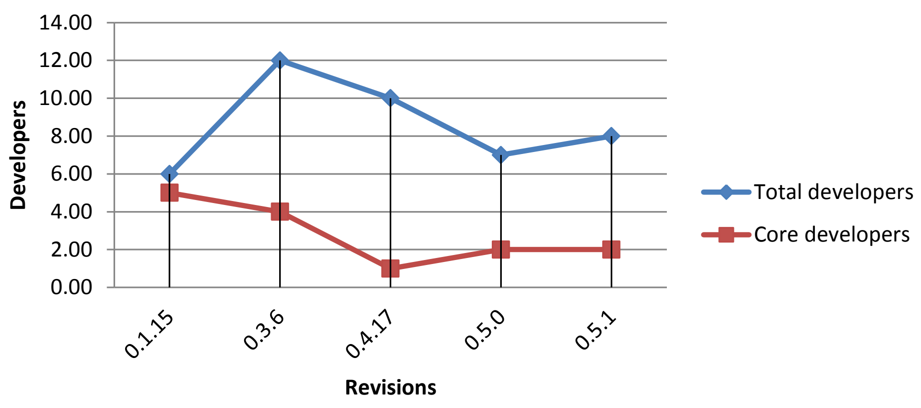

Figure 4.12: Total and core developers of Rascal.

It also can be mentioned that, as in terms of inner class density this project present a relative high value compared with the other studied cases (see Table 3.2), it should be known that this could potentially produce some deviation in the outcome of the coupling aggregation process, and consequently, on the definition of the core files.

4.2.3 Analysis

In this project the number of developers is smaller though in terms of productivity (SLOC) seems to be doing well (see Table 3.1).

What is particularly interesting in this case is the decrease of correlation in the last period. This is caused by a peripheral developer that is participating in a very high proportion. If this scenario were found in other studied cases, it could represent a threat to the definition of peripheral developers (as those who contribute less to the project [25]). But this is not the case and the situation is not replicated in other sampled systems.

It is important to mention that the number of both total and core developers, trough the life cycle of the project, tends to decrease (see Figure 4.12) and this may also have an impact in the drop of the correlation (as the number of core authors is reduced and they may be focused, intentionally, in other parts of the software). This fall in the number of individuals evolving the technical core could also be related with the stability reached. For example, it can be seen in Appendix A that the coupling levels of all the files of the core are (in general) either not increasing or decreasing. This may mean that the core has reached certain stability, and that the development could be focused in other activities (i.e. maintenance or refactoring) and that core developers could have moved to peripheral parts of the code or to different activities.

Although this project seems to be centralized in terms of development (with a small group of top contributors doing considerably more than the rest), another factor that may have influenced the drop in the correlation is that the project is, apparently, becoming slightly more decentralized (number of contributions are more equally distributed among developers, see Figure 4.11).

4.3 Clojure

4.3.1 Case Presentation

Clojure is a dynamic programming language predominantly functional that is aimed to offer both the features of a scripting language combined with an infrastructure for multithreading programming. Clojure is a dialect of Lisp that is targeted to the Java Virtual Machine and compiled into Bytecode [5]. According to the data mined from its main repository [6], the project count with 98 authors across the studied period (more than 6 years). Also, the project has several user groups in different countries [7].

4.3.2 Results

This project is also presenting a positive and very high correlation between core developers and top contributors. Its values are variating between 90% and 100% in the time period studied (see Figures 4.13 and 4.14).

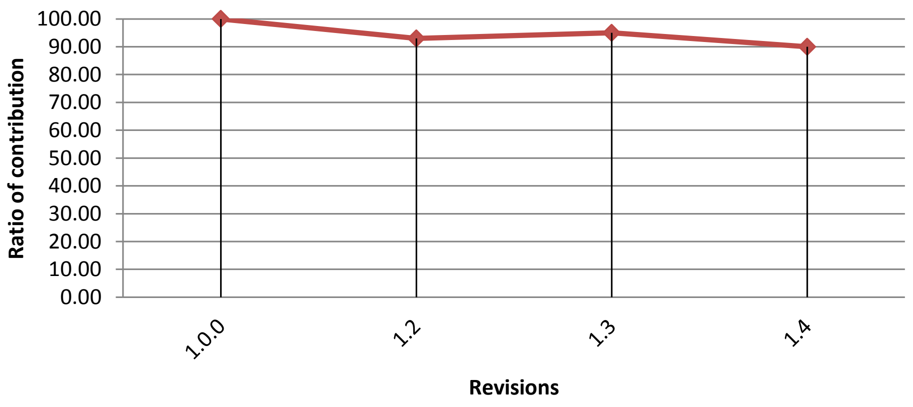

Figure 4.13: Ratio of contribution of core developers of Clojure (from total).

As it can be observed, in this project, there is a quite direct relation between core and top developers (see Figure 4.15).

[5]http://clojure.org/
[6]https://github.com/clojure/clojure
[7]http://dev.clojure.org/display/community/Clojure+User+Groups

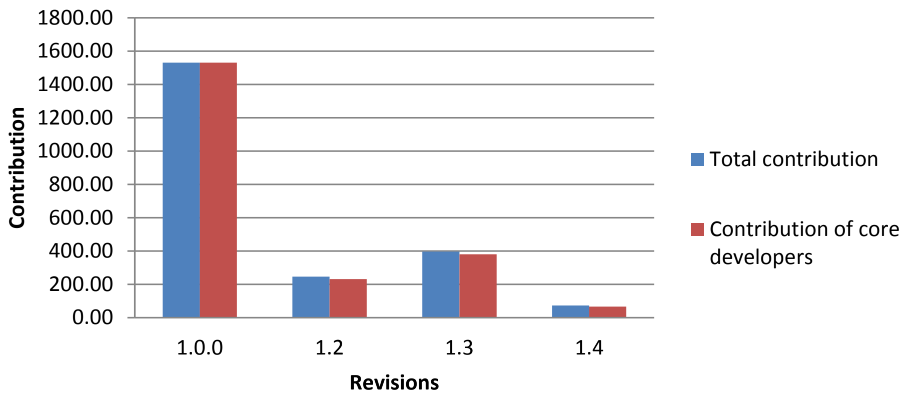

Figure 4.14: Total and core developer contribution of Clojure.

In this case, there is a great amount of contribution in the first time frame studied and it is produced by a single developer (see Figures 4.14 and 4.15). Also, throughout the evolution of the system there is a considerable growth in the number of contributors involved, especially in the periods between versions 1.0.0 and 1.3 (see Figure 4.15).

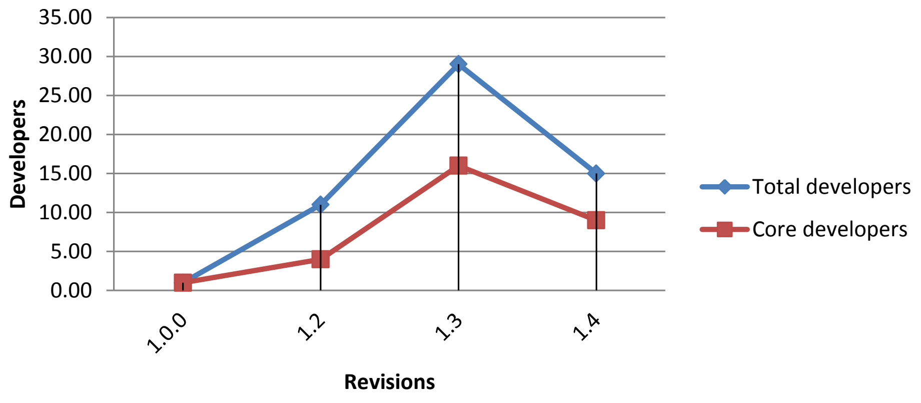

Figure 4.15: Total and core developers of Clojure.

It also can be appreciated that the core developers (and particularly those who contribute more to the core) produce the most of the system (see Figures 4.16 and 4.17).

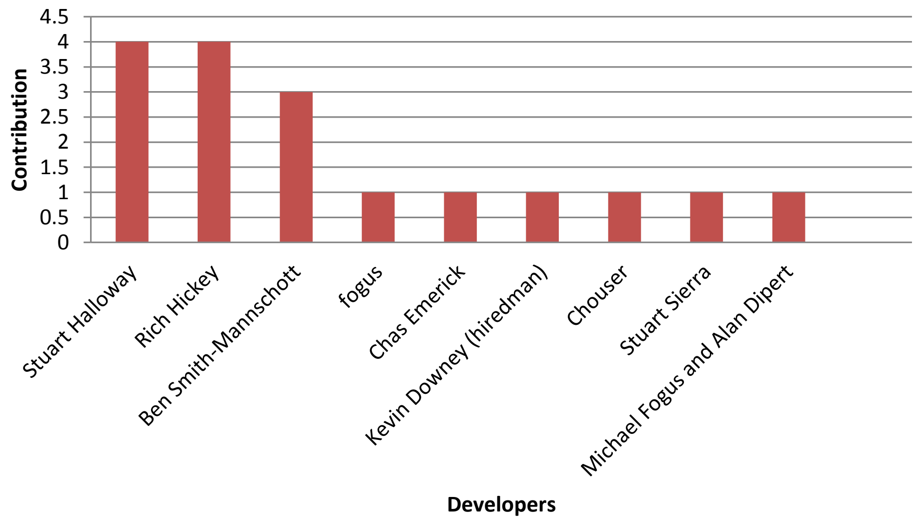

Figure 4.16: Contributions to the core of Clojure (in time frame preceding revision 1.4).

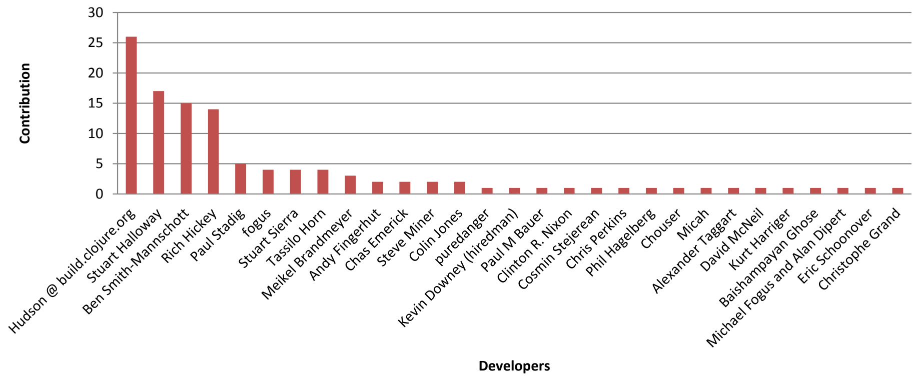

Figure 4.17: Total contributions to Clojure (time frame preceding revision 1.4).

4.3.3 Analysis

As it was shown (see Figure 4.17) this project is also centralized in terms of development, with a reduced number of contributors doing the most of the work.

In this system, it seems that a small group of developers are adopting the role of committers, while other larger group of people are participating as authors in most of the periods (see Appendix B). In this case, the authorship tracking feature provided by Git is being utilized as part of the development process. This is one of the few studied systems that shows this characteristic.

In this case it is interesting that core developers are highly dominant in terms of participation (see Figure 4.14) and that the increase in the number of authors (see Figure 4.15) is not being proportionately reflected in the amount of contribution. However, this consideration could have an opposite meaning because it should be considered that the period prior to version 1.0.0 is presenting disproportionate amount of contribution. This could be product to the fact that this situation is perceived in an also disproportionately long time frame (with more than three years out of a total of seven for all the revisions).

It is also interesting to highlight that the increase in the amount of developers is followed by a pronounced rise in the coupling levels as in the time frame between revisions 1.0.0 and 1.3, there is a sustained growth of both measures (see Figure 4.15 and Appendix A).

4.4 Oscar

4.4.1 Case Presentation

Oscar [8] is an open source EMR (electronic medical record) application. The project has been produced collaboratively by a group of developers and health care providers of Canada and it is mainly supported, evolved and promoted by a team at Mc Master university (OSCAR McMaster). According to the data available on its main repository [9] the project has been developed for almost 10 years and features a total of 52 software developers.

4.4.2 Results

Again, in this sample it can be appreciated that developers that are contributing more to the technical core are in general those who produce the most of the total system (see Figures 4.21 and 4.22). Consequently, the correlation core-top developer contribution is, as in the other cases, very high (between 70% and 98% approximately) in most of the time frames analyzed (see Figures 4.18 and 4.19). In the periods between revisions 2009-06-01 and 2009-08-13, a steep fall in correlation can be observed but, as it is clear in Figure 4.19 the cause of it is a drop in the total contribution, related to the studied structure, in both time frames.

[8]http://oscarmcmaster.org/

[9]git://oscarmcmaster.git.sourceforge.net/gitroot/oscarmcmaster/oscarmcmaster

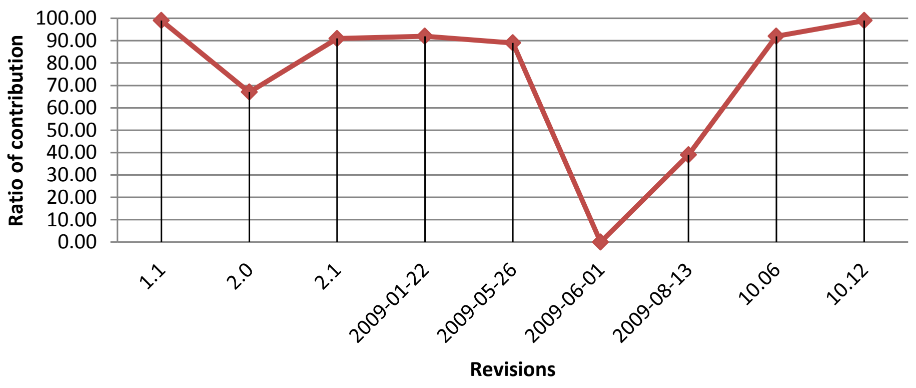

Figure 4.18: Ratio of contribution of core developers of Oscar (from total).

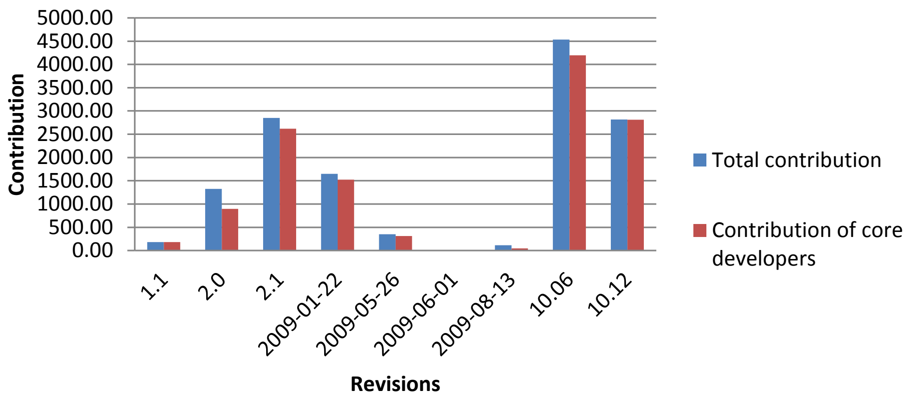

Figure 4.19: Total and core developer contribution of Oscar.

In terms of amount of authors, in this studied system there is also an almost proportional relation between core and top developers (see Figure 4.20). Core contributors are a fraction of the total, equivalently to the rest of samples.

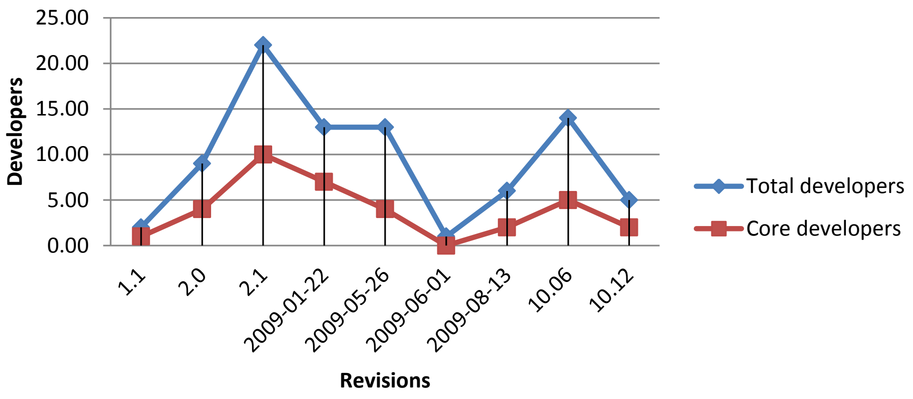

Figure 4.20: Total and core developers of Oscar.

In relation with the behaviour of specific developers inside and outside the technical core, it can be seen again, that most of those who access and work in the core are also producing the greatest amount of the total contribution in the last period (see Figures 4.21 and 4.22). This pattern is common in other revisions on the project.

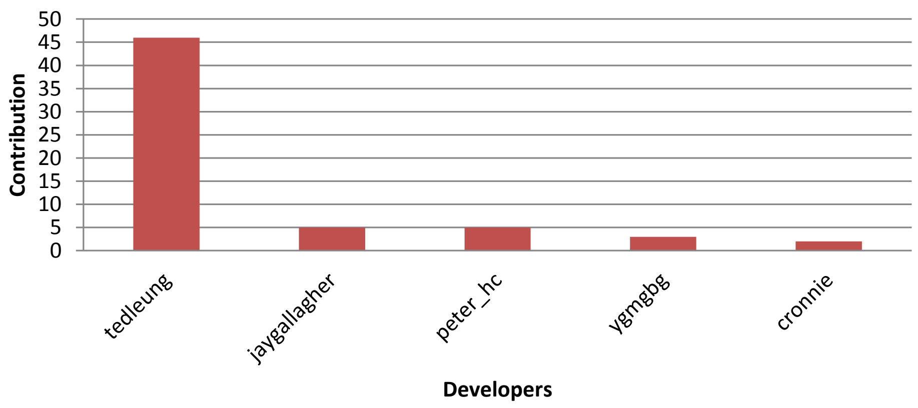

Figure 4.21: Contributions to the core of Oscar (in time frame preceding revision 10.12).

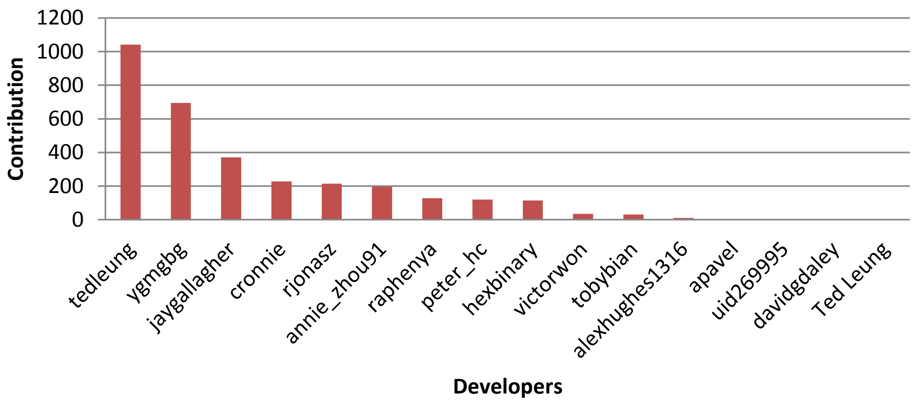

Figure 4.22: Total contributions to Oscar (time frame preceding revision 10.12).

4.4.3 Analysis

In this system, a fluctuation of the participation can be observed (Figure 4.19). It is also followed by a oscillating movement of the amount of participating developers (Figure 4.20). This could imply that this project is not evolving smoothly in terms of community (or that is experiencing other contingencies or particular stages in its evolution).

In the sense of the development distribution, this project can also be considered as highly centralized, as can be appreciated in Figure 4.22 (a reduced number of authors is doing more than most of the developers).

In this project, the authorship tracking feature is not part of the development process as there is no single difference in the number of those who are committers and authors (see Appendix B).

Oscar presents, in relation with other systems studied, lower levels of coupling in the core (see Appendix A). What is interesting is that, at the same time, this system is the largest in terms of lines of code and source files (see Tables 3.1 and 3.2). Consequently, these two factors combined may imply that the software structure is well modularized (though other measure would be required to conclude this).

4.5 Solr

4.5.1 Case Presentation

Solr is an open source enterprise search platform, part of the Apache Lucene project. It supplies distributed search, index replication and some of their features include full-text search, hit highlighting, faceted search, dynamic clustering, database integration, and rich document handling [10]. According to the data available in the studied repository of the project [11], it accounts with 17 developers.

4.5.2 Results

In this project it can be appreciated that the correlation between those who produce the core and those who most contribute is very high (between 90% and 100% for the studied time frames, see Figures 4.23 and 4.24).

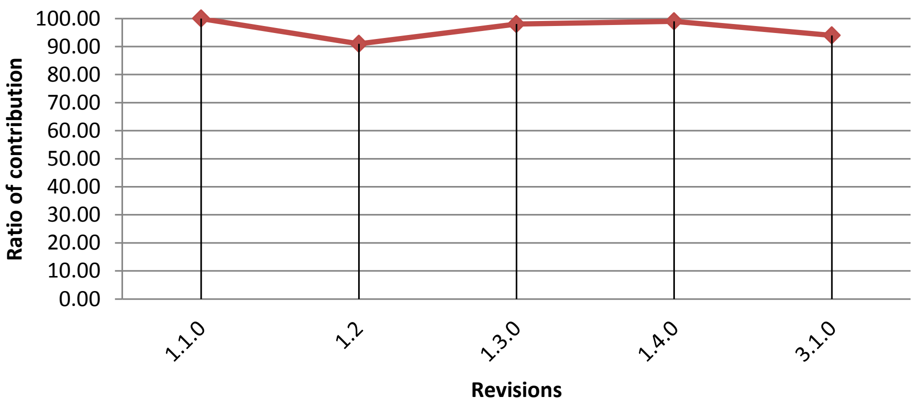

Figure 4.23: Ratio of contribution of core developers of Solr (from total).

[10]http://lucene.apache.org/solr/
[11]https://github.com/apache/lucene-solr

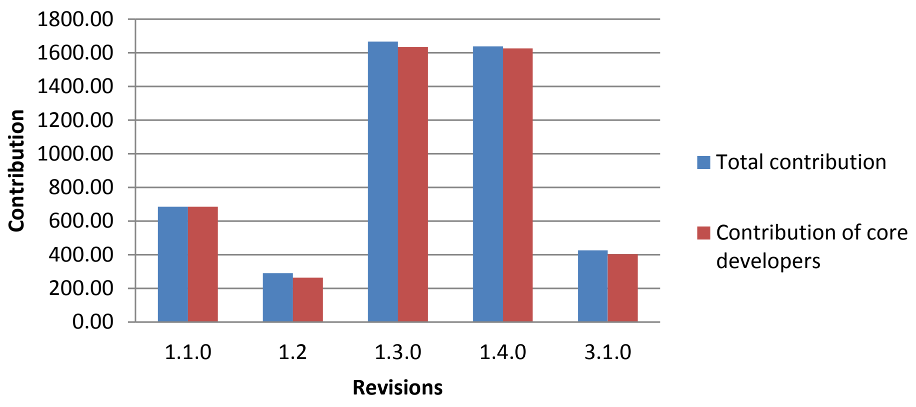

Figure 4.24: Total and core developer contribution of Solr.

Considering the number of contributors, in this case there is also a quite pronounced relationship between core and top developers (see Figure 4.25). What is different from other studies is that here the core group is greater in proportion to the group of total contributors.

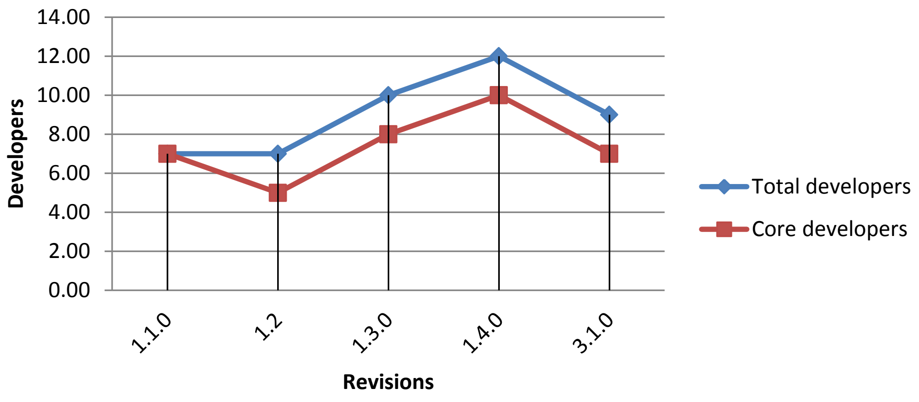

Figure 4.25: Total and core developers of Solr.

In terms of particular developers, it can be appreciated that the core contributors are (except in one case) the same who produce the greatest amount of total contribution (see Figures 4.26 and 4.27).

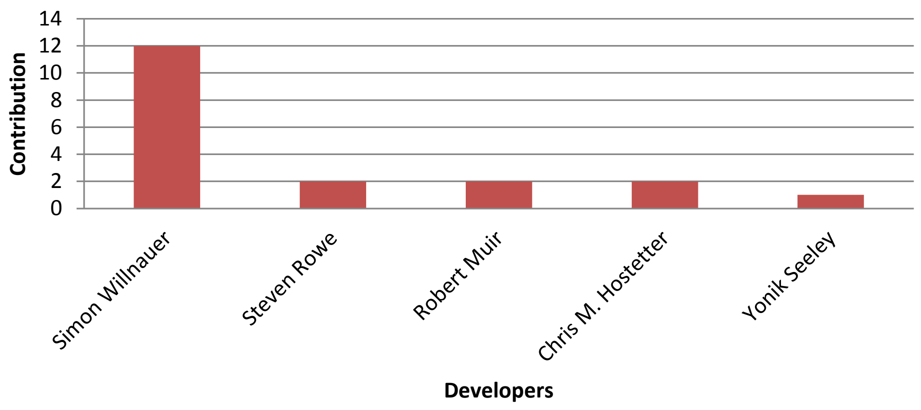

Figure 4.26: Contributions to the core of Solr (in time frame preceding revision 3.1.0).

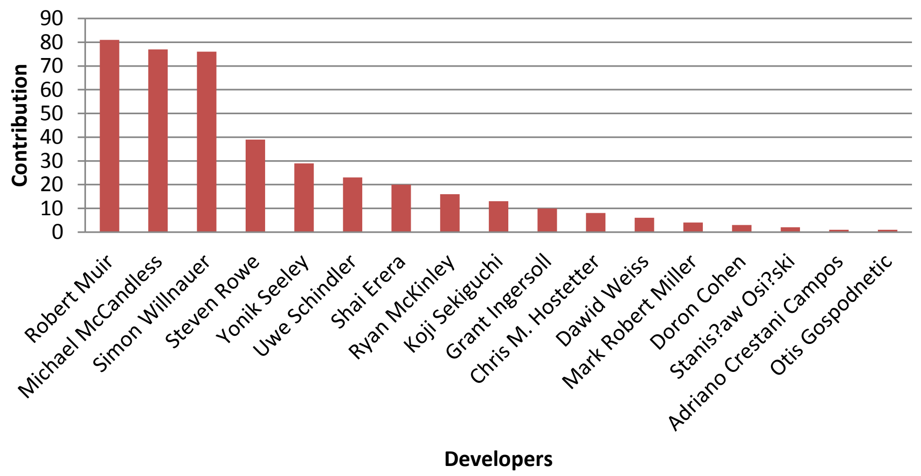

Figure 4.27: Total contributions to Solr (time frame preceding revision 0.3.1.0).

4.5.3 Analysis

Solr presents a highly centralized contribution pattern, as can be seen in Figure 4.27, where three developers are producing considerably more than the rest of the contributors of the project. In this case, there is no variation between the number of authors and developers despite the project shows a tendency to incorporate more contributors throughout the time (see Appendix B and Figure 4.25). It seems that there is no utilization of the authorship tracking feature provided by Git.

4.6 Voldemort

4.6.1 Case Presentation

Voldemort is an open source distributed key-value storage system. Some of their features include automatic data replication over multiple servers, automatic data partitioning, pluggable serialization, data items versioning, no central point of failure and support for pluggable data placement strategies [12]. According to the master repository [13], the project count with 61 developers through a period of almost 4 years.

4.6.2 Results

With reference to the correlation between top and core contributors, in this open source project, three different behaviours can be observed (see Figures 4.28 and 4.29). First, in the time frame between revisions 0.57.1 and 0.80, it presents high levels (between 74% and 99%). Second, from revision 0.80 to 0.81, it drops abruptly within 0% and 20%. This happens because, as can be seen in Figure 4.29, there is no activity related to the technical core in the period. Finally, in the time frame between revisions 0.90 and 0.96, the correlation is considerable again though a pronounced decrease from 98% to 52% can be appreciated.

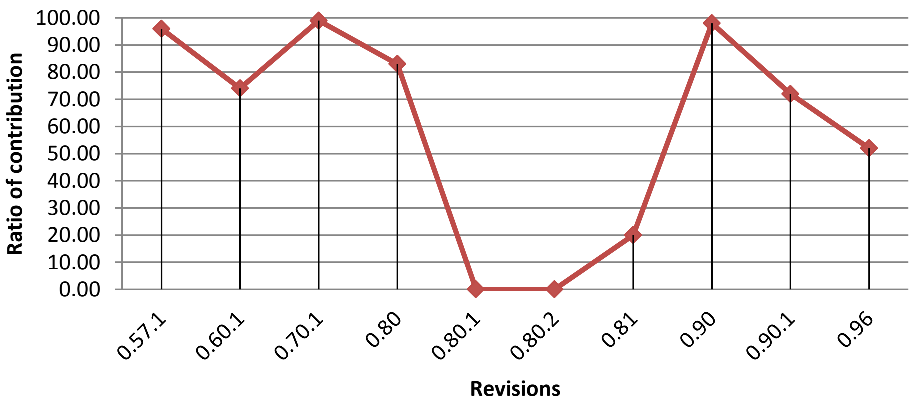

Figure 4.28: Ratio of contribution of core developers of Voldemort (from total).

[12]http://www.project-voldemort.com/voldemort/

[13]https://github.com/voldemort/voldemort

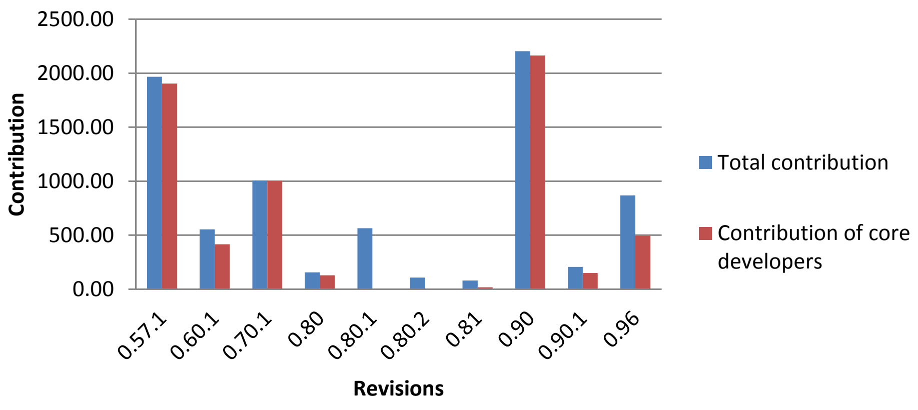

Figure 4.29: Total and core developer contribution of Voldemort.

In this case, there is an oscillating behavior in the number of total developers in each time frame analyzed (Figure 4.30). It can be observed that the peaks are reached in the first and in the last studied periods (prior to versions 0.57.1 and 0.96 respectively). Core developers are still a fraction of the total but in a period (prior to version 0.70.1) they reach almost the total number of developers.

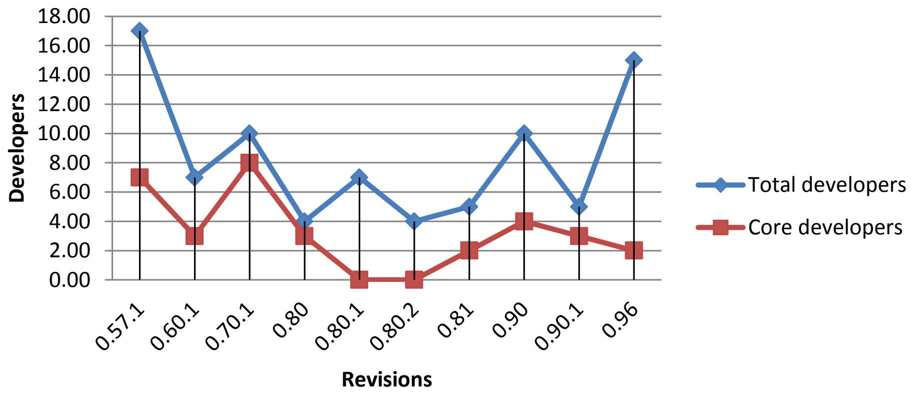

Figure 4.30: Total and core developers of Voldemort.

In this sample it can be seen that the two core developers are amongst the top three contributors of the time frame analyzed (see Figures 4.31 and 4.32). Again, this tendency can be observed in all the periods (provided that there are some activity related to the technical core).

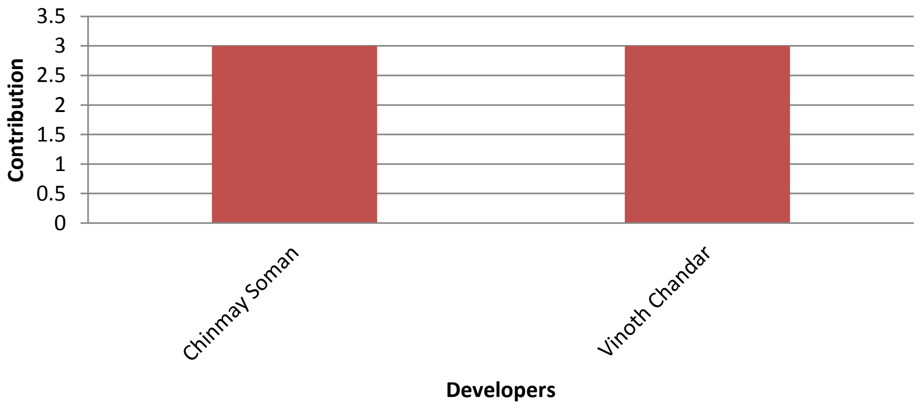

Figure 4.31: Contributions to the core of Voldemort (in time frame preceding revision 0.96).

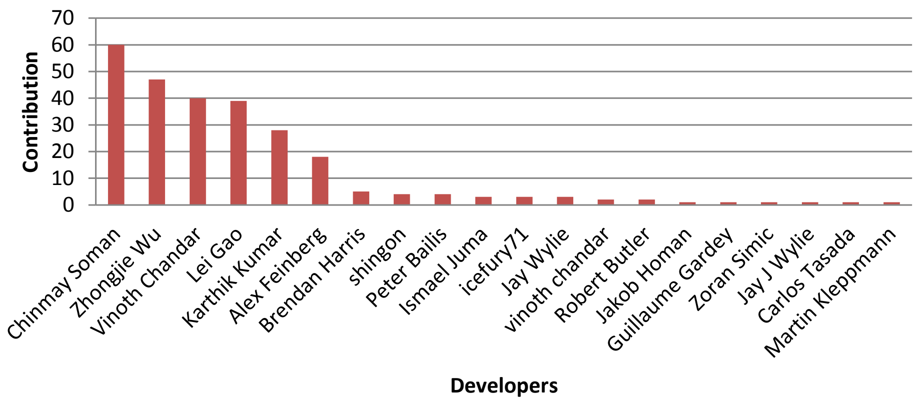

Figure 4.32: Total contributions to Voldemort (time frame preceding revision 0.96).

Finally, as it can be appreciated in Appendix A, the core is generally composed by two files. One of them (*VAdminProto.java*) is showing an increasing coupling behavior, while the other, remains relatively stable throughout the time (*VProto.java*).

4.6.3 Analysis

As in the previous analyzed cases, it is perceived that this project also presents development centralization characteristics, with a reduced group of people doing considerable more than most of the team (see Figure 4.32).

In terms of community, the project seems to be unstable, because the number of developers is fluctuating (see Figure 4.30). This may mean that the interest or commitment to the project are not sustained in time. In this sense it also can be observed that in some periods the activity in the core falls to zero and this may be product of a lack of interest or due to the migration of core developers to peripheral parts of the code. Also, another potential indication of the instability in relation to the community, is the fluctuating number of core developers (see Figure 4.30).

Considering the productivity in this project, it can be observed that it is not following a sustained tendency (see Figure 4.29) but it is interesting that the relation between the number of developers and the amount of participation seems to be, in general, positively related (in revisions 0.57.1, 0.70.1, 0.90.1 and 0.96, see Figures 4.29 and 4.30).

In this case, in most of the time frames studied, it can be perceived a considerable difference in the number of committers and authors (see Appendix B). This is reflecting that the authorship tracking feature provided by Git is utilized as part of the process.

Chapter 5

Conclusions

In previous publications core developers were defined as those who produce or modify the core of the systems. The necessity to validate this definition was suggested [1] and it was also perceived that at the moment it had not been made with a wide extent of generality [22]. In the present study it was found that, for the sample of open source systems studied, the group of developers that have access, produce and modify the part of the systems that present high levels of coupling (core developers), are also those who participate more actively and contribute the most to these systems (top contributors).

Considering the developers individually, it was found that, in general terms, those who produce the core in a great extent are also the top contributors of the project.

Also, it was validated that development centralization, as it was mentioned in the related work [25, 20, 10], is an important characteristic of open source projects. Even the projects with a reduced number of developers (which the contribution, knowledge and commitment levels are expected to be similar) were found to be just slightly more decentralized.

Simultaneously, and for all the sampled open source projects, it was observed that the number of artifacts that present core properties is reduced in relation to the total.

Additionally, it was observed that the authorship tracking option present in the Git version control system is not utilized in most of the sampled projects. Though it is not clear if this is product of an intention or an omission, it would be helpful for this type of research if this feature could be exploited.

Chapter 6

Future Work

First, it would be interesting to understand variations product of defining the core in different ways, studying other variables that can influence the meaning of technical core. For example, outbound coupling as it is presented in [26], and architectural analysis (i.e. extension points, interfaces and architectural patterns used in open source systems).
Second, it would be interesting to have a different sampling strategy in the sense of focusing on open source projects that are in earlier stages of their evolution and that seem not to be doing well in terms of health [1], although in these cases, it would be difficult to obtain adequate data sets.
Finally, as it was shown that core developers are correlated with those who are top contributors, it would be interesting to have an inverse approach. That is, finding clusters of top developers and looking for correlations between them and the properties of the software.

Bibliography

[1] Chintan Amrit and Jos van Hillegersberg. Exploring the impact of socio-technical core-periphery structures in open source software development. *J Inf technol*, 25:216–229, 06 2010.

[2] Carliss Y. Baldwin and Kim B. Clark. The architecture of participation: Does code architecture mitigate free riding in the open source development model? *Management Science*, 52(7):1116–1127, July 2006.

[3] C. Bird, P.C. Rigby, E.T. Barr, D.J. Hamilton, D.M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1 –10, may 2009.

[4] Stephen P Borgatti and Martin G Everett. Models of core/periphery structures. *Social Networks*, 21(4):375 – 395, 2000.

[5] L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1):91 –121, jan/feb 1999.

[6] Scott Chacon. *Pro Git*. Apress, 2009.

[7] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476 –493, jun 1994.

[8] M.E. Conway. How do committees invent? *Datamation*, Vol. 14(No. 4):28–31, 1968.

[9] K. Crowston, Kangning Wei, Qing Li, and J. Howison. Core and periphery in free/libre and open source software team communications. In *System Sciences, 2006. HICSS '06. Proceedings of the 39th Annual Hawaii International Conference on*, volume 6, page 118a, jan. 2006.

[10] Kevin Crowston and James Howison. The social structure of free and open source software development. *First Monday*, 10(2 - 7), February 2005.

[11] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/libre open-source software development: What we know and what we do not know. *ACM Comput. Surv.*, 44(2):7:1–7:35, March 2008.

[12] Daniel A. Hojman and Adam Szeidl. Core and periphery in networks. *Journal of Economic Theory*, 139(1):295 – 309, 2008.

[13] Shih-Kun Huang and Kang-min Liu. Mining version histories to verify the learning process of legitimate peripheral participants. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[14] F.P. Brooks Jr. The mythical man-month. Essays on Software Engineering, Reading, MA, 1975. Addison-Wesley.

[15] Marian Jureczko and Diomidis Spinellis. Using object-oriented design metrics to predict software defects. In *Models and Methodology of System Dependability. Proceedings of RELCOMEX 2010: Fifth International Conference on Dependability of Computer Systems DepCoS*, Monographs of System Dependability, pages 69–81, Wrocław, Poland, 2010. Oficyna Wydawnicza Politechniki Wrocławskiej.

[16] M.J. LaMantia, Yuanfang Cai, A.D. MacCormack, and J. Rusnak. Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, pages 83 –92, feb. 2008.

[17] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM.

[18] Alan MacCormack, Carliss Baldwin, and John Rusnak. The architecture of complex systems: Do core-periphery structures dominate? *MIT Sloan School of Management Working Paper*, pages 4770–10, 01 2010.

[19] Robert Martin. Object oriented design quality metrics - an analysis of dependencies. In *Proc. of Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA*, may 1994.

[20] 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, July 2002.

[21] J. Nonnen and P. Imhoff. Identifying knowledge divergence by vocabulary monitoring in software projects. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 441 –446, march 2012.

[22] G.A. Oliva, F.W. Santana, K.C.M. de Oliveira, C.R.B. de Souza, and M.A. Gerosa. Characterizing key developers: A case study with apache ant. 2011.

[23] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115 –139, 1974.

[24] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, page 10, may 2007.

[25] 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.

[26] Andy Zaidman and Serge Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):387–417, 2008.

Appendix A

Core Java Files of the Open Source Systems

In this section, all the Java files that are part of the technical core in each time frame studied are presented. Following the name of the file, the coupling values for each computed metric are presented. In order to interpret the data the following convention must be used: *file name*, *CBO*, *RFC*, *Ca*, *Ce*, *Ibound dependencies*, *Outbound dependencies*.

Jenkins

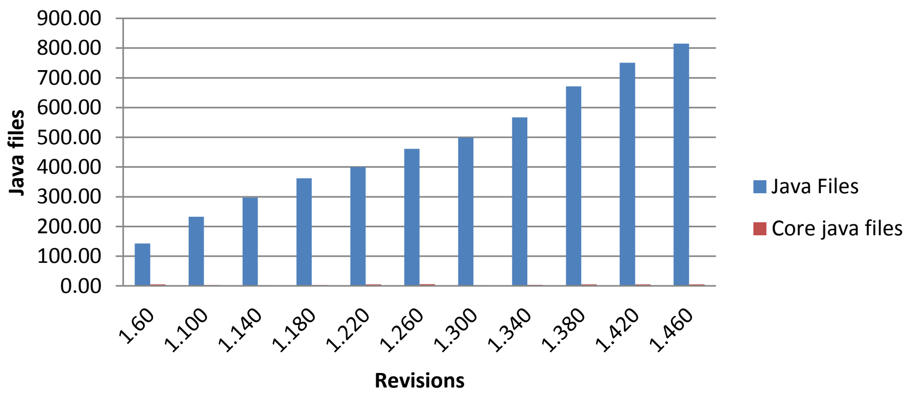

Figure A.1: Total and core Java files of Jenkins.

Revision 1.60

hudson/model/Project.java,48,186,48,48,7,15
hudson/model/Build.java,50,143,50,50,8,14
hudson/model/Hudson.java,63,306,45,63,7,25
hudson/scm/CVSSCM.java,70,293,16,70,7,25
hudson/model/Run.java,44,226,51,44,10,30

Revision 1.100

hudson/scm/SubversionSCM.java,167,342,51,167,13,57
hudson/FilePath.java,134,340,134,134,59,88
hudson/scm/CVSSCM.java,129,487,36,129,14,61

Revision 1.140

hudson/scm/SubversionSCM.java,198,449,67,198,17,66
hudson/FilePath.java,143,394,151,143,66,101

Revision 1.180

hudson/scm/SubversionSCM.java,207,481,67,207,17,66
hudson/FilePath.java,149,413,154,149,68,104
hudson/model/AbstractBuild.java,73,216,100,73,23,25

Revision 1.220

hudson/model/Hudson.java,198,644,131,198,14,61
hudson/scm/SubversionSCM.java,222,534,74,222,17,74
hudson/FilePath.java,157,432,157,157,68,108
hudson/model/Queue.java,60,203,58,60,16,29
hudson/model/AbstractBuild.java,77,224,112,77,23,26

Revision 1.260

hudson/model/Hudson.java,219,704,151,219,16,68
hudson/scm/SubversionSCM.java,237,541,75,237,17,74
hudson/FilePath.java,165,465,175,165,70,111
hudson/model/Queue.java,77,250,71,77,17,33
hudson/model/UpdateCenter.java,83,243,52,83,28,46
hudson/model/AbstractBuild.java,80,228,133,80,26,26

Revision 1.300

hudson/FilePath.java,205,568,206,205,87,132
hudson/model/UpdateCenter.java,95,299,58,95,27,49

Revision 1.340

hudson/model/Hudson.java,299,944,281,299,21,101
hudson/FilePath.java,286,749,265,286,102,178
hudson/model/Queue.java,148,436,130,148,25,57
hudson/util/ProcessTree.java,84,335,71,84,22,43

Revision 1.380

hudson/model/Hudson.java,323,986,319,323,23,105
hudson/FilePath.java,245,648,236,245,95,150
hudson/model/Queue.java,139,419,122,139,26,48

hudson/util/ProcessTree.java,128,403,93,128,28,64
hudson/model/UpdateCenter.java,120,377,64,120,26,55

Revision 1.420

jenkins/model/Jenkins.java,345,982,354,345,27,113
hudson/FilePath.java,245,661,238,245,95,151
hudson/model/Queue.java,145,430,128,145,26,48
hudson/util/ProcessTree.java,128,408,93,128,28,64
hudson/model/UpdateCenter.java,145,407,85,145,38,68

Revision 1.460

jenkins/model/Jenkins.java,360,1023,386,360,28,117
hudson/FilePath.java,262,708,246,262,101,163
hudson/model/Queue.java,150,446,132,150,26,48
hudson/util/ProcessTree.java,128,408,93,128,28,64
hudson/model/UpdateCenter.java,158,432,90,158,40,73

Rascal

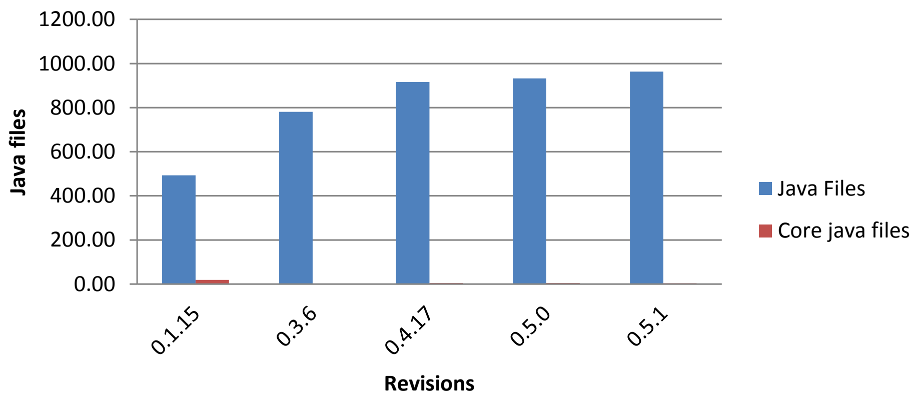

Figure A.2: Total and core Java files of Rascal.

Revision 0.1.15

org/rascalmpl/ast/Declaration.java,73,176,81,73,14,46
org/rascalmpl/ast/ImportedModule.java,28,56,32,28,7,15
org/rascalmpl/ast/Kind.java,35,69,58,35,12,13
org/rascalmpl/interpreter/Evaluator.java,428,1064,47,428,16,37
org/rascalmpl/ast/Assignable.java,35,94,61,35,9,18
org/rascalmpl/ast/Statement.java,121,322,243,121,32,65
org/rascalmpl/ast/Expression.java,230,691,733,230,104,104
org/rascalmpl/ast/CharClass.java,25,72,40,25,8,10
org/rascalmpl/ast/Assignment.java,26,51,46,26,9,10
org/rascalmpl/ast/ShellCommand.java,42,84,66,42,12,18

```
org/rascalmpl/test/TestFramework.java,26,69,43,26,42,6
org/rascalmpl/ast/Literal.java,40,80,68,40,9,17
org/rascalmpl/ast/Symbol.java,51,132,69,51,14,22
org/rascalmpl/ast/StringTemplate.java,29,110,44,29,9,20
org/rascalmpl/ast/Command.java,33,65,45,33,7,15
org/rascalmpl/ast/StringLiteral.java,26,52,36,26,6,13
org/rascalmpl/interpreter/result/Result.java,37,245,86,37,12,6
org/rascalmpl/ast/BasicType.java,74,147,179,74,28,26
org/rascalmpl/ast/Type.java,43,90,95,43,27,18
```

Revision 0.3.6

```
org/rascalmpl/ast/Statement.java,127,327,234,127,61,65
org/rascalmpl/ast/Sym.java,97,253,116,97,32,39
org/rascalmpl/ast/Expression.java,240,709,660,240,175,106
```

Revision 0.4.17

```
org/rascalmpl/ast/Statement.java,129,328,205,129,62,66
org/rascalmpl/ast/Sym.java,99,254,90,99,32,40
org/rascalmpl/ast/Expression.java,255,758,613,255,184,110
```

Revision 0.5.0

```
org/rascalmpl/interpreter/Evaluator.java,105,387,376,105,18,26
org/rascalmpl/ast/Statement.java,129,328,205,129,62,66
org/rascalmpl/ast/Sym.java,99,254,90,99,32,40
org/rascalmpl/ast/Expression.java,255,758,615,255,184,110
```

Revision 0.5.1

```
org/rascalmpl/ast/Statement.java,129,328,152,129,62,66
org/rascalmpl/ast/Sym.java,103,264,93,103,33,42
org/rascalmpl/ast/Expression.java,255,758,490,255,184,110
```

Clojure

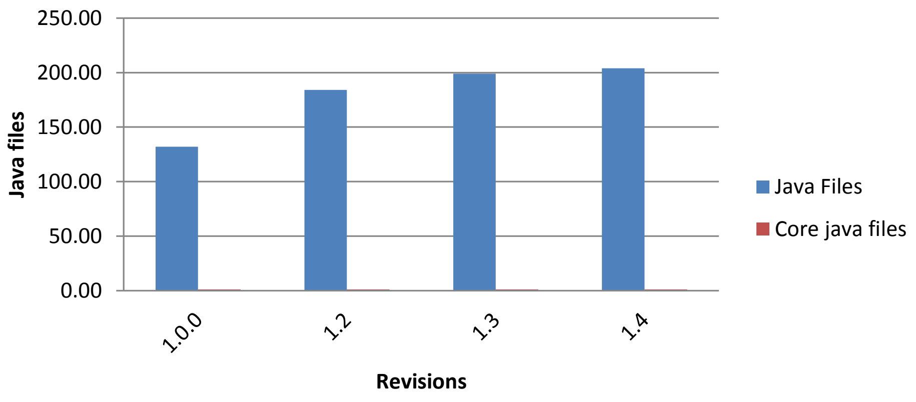

Figure A.3: Total and core Java files of Clojure.

Revision 1.0.0

clojure/lang/Compiler.java,666,1487,384,666,78,208

Revision 1.2

clojure/lang/Compiler.java,904,2116,545,904,103,266

Revision 1.3

clojure/lang/Compiler.java,1000,2442,589,1000,106,285

Revision 1.4

clojure/lang/Compiler.java,1001,2453,593,1001,106,285

Oscar

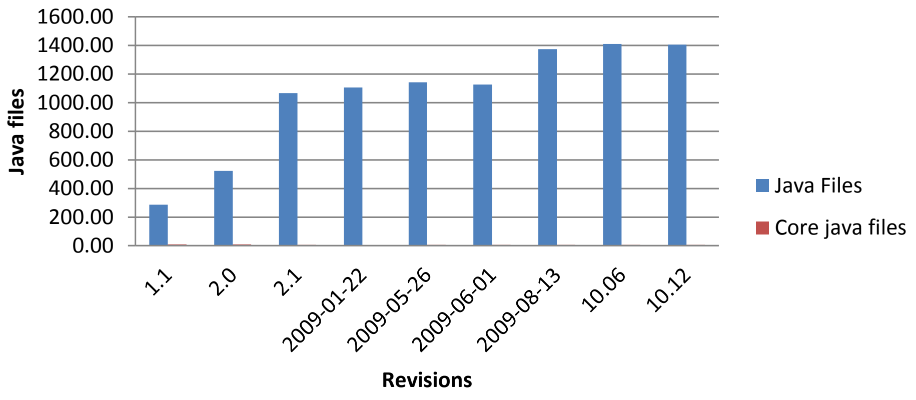

Figure A.4: Total and core Java files of Oscar.

Revision 1.1

oscar/oscarRx/data/RxAllergyData.java,10,61,14,10,5,7
oscar/oscarRx/data/RxCodesData.java,6,40,6,6,2,7
oscar/oscarRx/data/RxPatientData.java,18,134,15,18,3,13
oscar/form/data/FrmData.java,6,35,4,6,2,7
oscar/oscarRx/data/RxInteractionsData.java,4,29,4,4,2,4
oscar/oscarRx/data/RxDrugData.java,6,73,9,6,2,4
oscar/oscarRx/data/RxPrescriptionData.java,13,228,24,13,2,9
oscar/oscarEncounter/data/EctFormData.java,6,34,4,6,2,7
oscar/oscarReport/data/RptConsultReportData.java,9,59,6,9,3,14
oscar/oscarDB/DBHandler.java,3,22,99,3,2,4

Revision 2.0

oscar/oscarRx/data/RxAllergyData.java,5,36,11,5,4,7
oscar/billing/model/Appointment.java,7,66,9,7,5,6
oscar/billing/cad/model/CadProcedimentos.java,3,23,13,3,4,6
oscar/oscarRx/data/RxPatientData.java,18,137,13,18,3,13
oscar/oscarRx/data/RxDrugData.java,22,107,28,22,17,44
oscar/oscarRx/data/RxPrescriptionData.java,17,284,25,17,2,8
oscar/util/DAO.java,3,25,9,3,9,5
oscar/oscarDB/DBHandler.java,3,22,186,3,6,4
oscar/oscarBilling/data/BillingFormData.java,13,64,14,13,6,19

Revision 2.1

oscar/oscarDemographic/data/DemographicData.java,22,198,64,22,6,18
oscar/oscarEncounter/pageUtil/NavBarDisplayDAO.java,8,87,37,8,3,14
oscar/oscarRx/data/RxPatientData.java,18,147,33,18,4,13
oscar/oscarBilling/ca/bc/data/BillingFormData.java,16,94,17,16,6,19

oscar/oscarRx/data/RxDrugData.java,26,126,37,26,18,48
oscar/oscarRx/data/RxPrescriptionData.java,21,345,38,21,2,8

Revision 2009-01-22

oscar/oscarDemographic/data/DemographicData.java,22,210,68,22,6,18
oscar/oscarEncounter/pageUtil/NavBarDisplayDAO.java,8,86,37,8,3,14
oscar/oscarRx/data/RxPatientData.java,18,149,36,18,4,13
oscar/oscarBilling/ca/bc/data/BillingFormData.java,16,96,17,16,6,19
oscar/oscarRx/data/RxDrugData.java,26,126,38,26,18,48

Revision 2009-05-26

oscar/oscarDemographic/data/DemographicData.java,22,213,74,22,6,18
oscar/oscarEncounter/pageUtil/EctDisplayAction.java,7,29,15,7,15,3
oscar/oscarEncounter/pageUtil/NavBarDisplayDAO.java,8,86,39,8,3,14
oscar/oscarRx/data/RxPatientData.java,18,153,39,18,4,13
oscar/oscarBilling/ca/bc/data/BillingFormData.java,16,96,17,16,6,19
oscar/oscarBilling/ca/bc/MSP/MSPReconcile.java,18,266,17,18,3,14
oscar/oscarRx/data/RxDrugData.java,26,128,40,26,18,48

Revision 2009-06-01

oscar/oscarDemographic/data/DemographicData.java,22,212,72,22,6,18
oscar/oscarEncounter/pageUtil/EctDisplayAction.java,7,29,15,7,15,3
oscar/oscarEncounter/pageUtil/NavBarDisplayDAO.java,8,86,39,8,3,14
oscar/oscarRx/data/RxPatientData.java,18,151,39,18,4,13
oscar/oscarBilling/ca/bc/data/BillingFormData.java,16,96,17,16,6,19
oscar/oscarBilling/ca/bc/MSP/MSPReconcile.java,18,266,17,18,3,14
oscar/oscarRx/data/RxDrugData.java,26,128,38,26,18,48

Revision 2009-08-13

oscar/oscarDemographic/data/DemographicData.java,24,220,76,24,6,19
oscar/oscarEncounter/pageUtil/NavBarDisplayDAO.java,8,86,41,8,3,14
oscar/oscarRx/data/RxPatientData.java,18,155,40,18,4,13
oscar/oscarBilling/ca/bc/data/BillingFormData.java,16,96,17,16,6,19
oscar/oscarBilling/ca/bc/MSP/MSPReconcile.java,18,266,17,18,3,14
oscar/oscarRx/data/RxDrugData.java,26,128,40,26,18,48

Revision 10.06

oscar/oscarDemographic/data/DemographicData.java,25,219,74,25,6,19
oscar/oscarEncounter/pageUtil/NavBarDisplayDAO.java,8,86,41,8,3,14
oscar/oscarRx/data/RxPatientData.java,18,151,40,18,4,13
oscar/oscarBilling/ca/bc/data/BillingFormData.java,19,101,16,19,6,20
oscar/oscarBilling/ca/bc/MSP/MSPReconcile.java,18,263,17,18,3,14
oscar/oscarRx/data/RxDrugData.java,26,132,42,26,18,48

Revision 10.12

oscar/oscarDemographic/data/DemographicData.java,35,281,75,35,6,19
oscar/oscarEncounter/pageUtil/NavBarDisplayDAO.java,10,87,47,10,3,14
oscar/oscarRx/data/RxPatientData.java,24,147,40,24,4,13
oscar/oscarBilling/ca/bc/data/BillingFormData.java,19,100,16,19,6,20
oscar/oscarBilling/ca/bc/MSP/MSPReconcile.java,23,258,17,23,3,14
oscar/oscarRx/data/RxDrugData.java,32,135,42,32,18,48

Solr

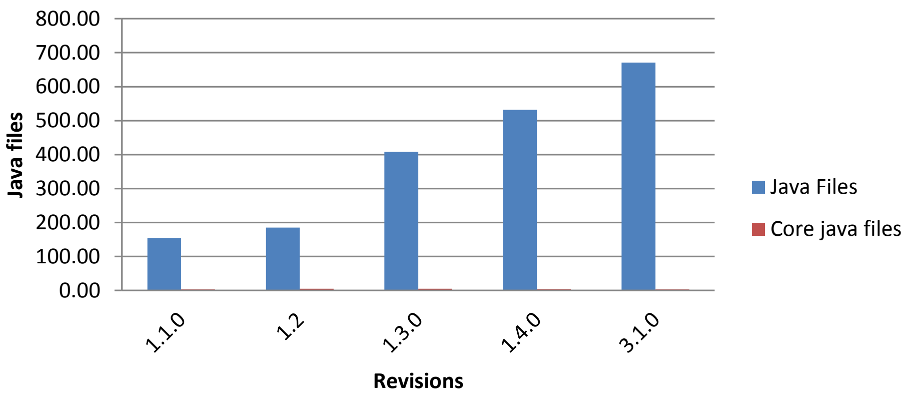

Figure A.5: Total and core Java files of Solr.

Revision 1.1.0

org/apache/solr/schema/IndexSchema.java,33,164,49,33,15,24
org/apache/solr/search/SolrIndexSearcher.java,108,257,52,108,20,49
org/apache/solr/core/SolrCore.java,61,205,36,61,10,31

Revision 1.2

org/apache/solr/servlet/SolrRequestParsers.java,54,115,19,54,10,23
org/apache/solr/schema/IndexSchema.java,41,199,59,41,18,29
org/apache/solr/search/SolrIndexSearcher.java,111,264,53,111,18,50
org/apache/solr/handler/CSVRequestHandler.java,60,111,23,60,11,26
org/apache/solr/core/SolrCore.java,54,170,45,54,12,30

Revision 1.3.0

org/apache/solr/client/solrj/impl/XMLResponseParser.java,48,129,40,48,12,17
org/apache/solr/schema/IndexSchema.java,72,285,97,72,27,43
org/apache/solr/search/ValueSourceParser.java,132,156,40,132,19,25
org/apache/solr/search/SolrIndexSearcher.java,117,337,77,117,20,54
org/apache/solr/core/SolrCore.java,83,309,90,83,17,44

Revision 1.4.0

org/apache/solr/schema/IndexSchema.java,82,311,110,82,30,47
org/apache/solr/search/ValueSourceParser.java,190,244,57,190,27,40
org/apache/solr/search/SolrIndexSearcher.java,107,318,76,107,14,36
org/apache/solr/core/SolrCore.java,97,345,107,97,21,46

Revision 3.1.0

org/apache/solr/schema/IndexSchema.java,92,341,119,92,31,48
org/apache/solr/search/ValueSourceParser.java,327,505,132,327,63,79
org/apache/solr/core/SolrCore.java,99,347,116,99,22,45

Voldemort

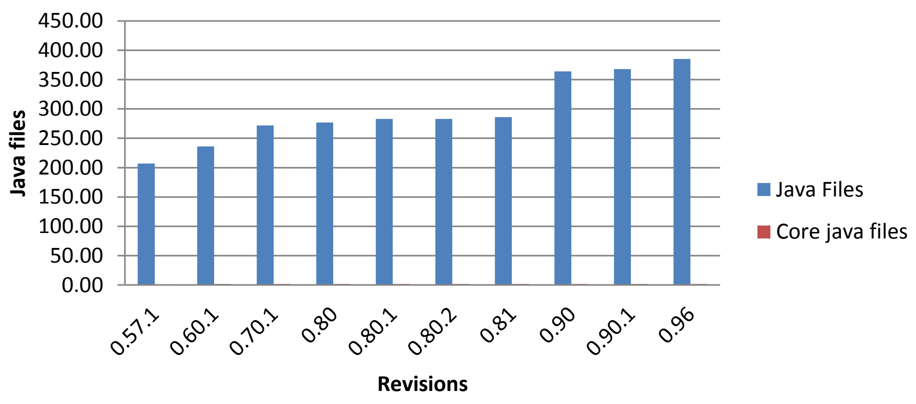

Figure A.6: Total and core Java files of Voldemort.

Revision 0.57.1

voldemort/client/protocol/pb/VProto.java,606,2077,194,606,28,82

Revision 0.60.1

voldemort/client/protocol/pb/VAdminProto.java,656,2374,189,656,30,96
voldemort/client/protocol/pb/VProto.java,606,2077,231,606,37,82

Revision 0.70.1

voldemort/client/protocol/pb/VAdminProto.java,922,3353,260,922,41,127
voldemort/client/protocol/pb/VProto.java,606,2092,240,606,40,82

Revision 0.80

voldemort/client/protocol/pb/VAdminProto.java,998,3591,291,998,47,136
voldemort/client/protocol/pb/VProto.java,606,2092,247,606,41,82

Revision 0.80.1

voldemort/client/protocol/pb/VAdminProto.java,998,3591,291,998,47,136
voldemort/client/protocol/pb/VProto.java,606,2092,247,606,41,82

Revision 0.80.2

voldemort/client/protocol/pb/VAdminProto.java,998,3591,291,998,47,136
voldemort/client/protocol/pb/VProto.java,606,2092,247,606,41,82

Revision 0.81

voldemort/client/protocol/pb/VAdminProto.java,998,3603,291,998,47,136
voldemort/client/protocol/pb/VProto.java,606,2092,247,606,41,82

Revision 0.90

voldemort/client/protocol/pb/VAdminProto.java,1922,7169,569,1922,87,258
voldemort/client/protocol/pb/VProto.java,629,2272,296,629,53,86

Revision 0.90.1

voldemort/client/protocol/pb/VAdminProto.java,1996,7408,587,1996,90,267
voldemort/client/protocol/pb/VProto.java,629,2272,299,629,54,86

Revision 0.96

voldemort/client/protocol/pb/VAdminProto.java,2221,8233,652,2221,100,295
voldemort/client/protocol/pb/VProto.java,629,2272,305,629,56,86

Appendix B

Commiters and Authors

In this research work, it was found that the authorship tracking mechanism offered by Git is utilized in different extent. In this section, the results related with the measurement of the utilization of this feature are presented for each sampled system.

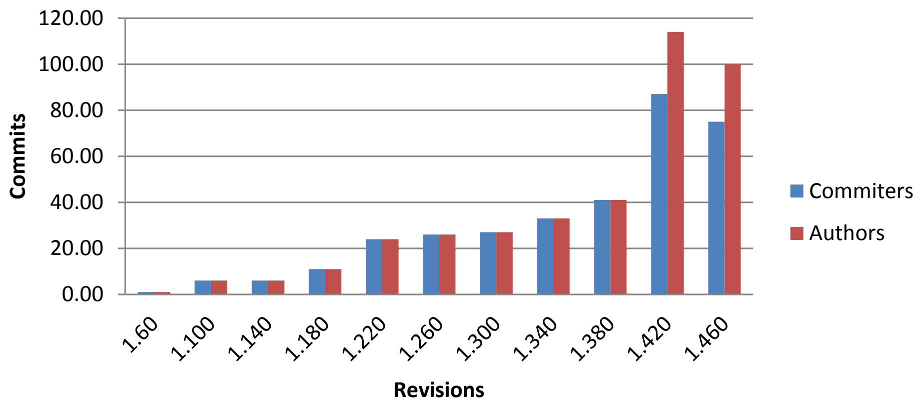

Figure B.1: Commits of commiters and authors of Jenkins.

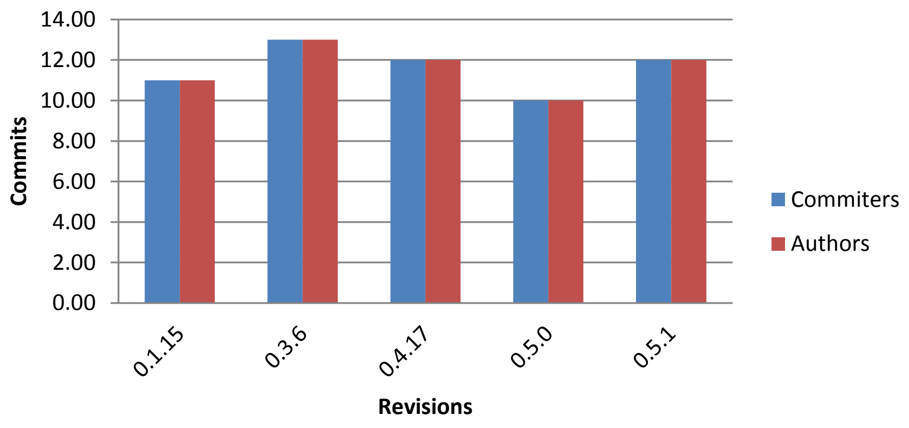

Figure B.2: Commits of commiters and authors of Rascal.

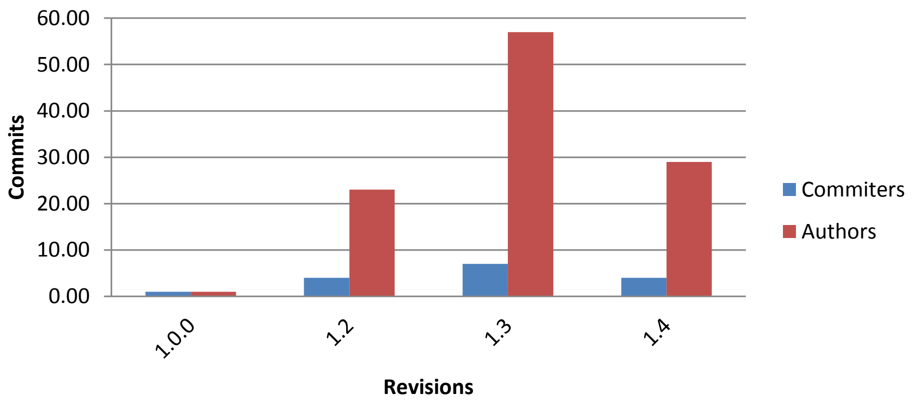

Figure B.3: Commits of commiters and authors of Clojure.

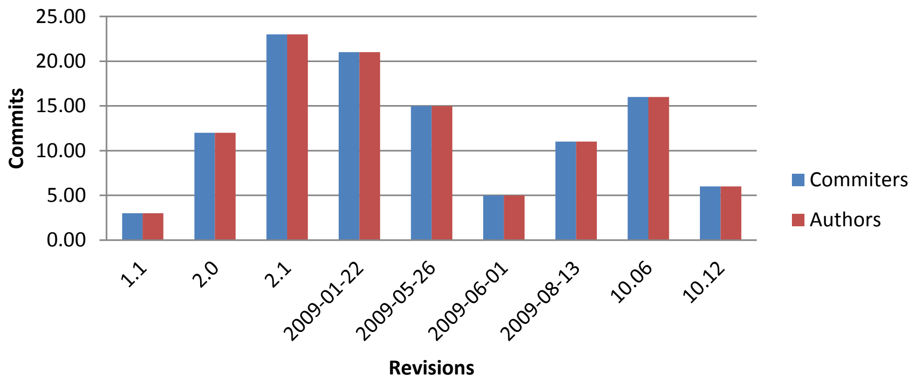

Figure B.4: Commits of commiters and authors of Oscar.

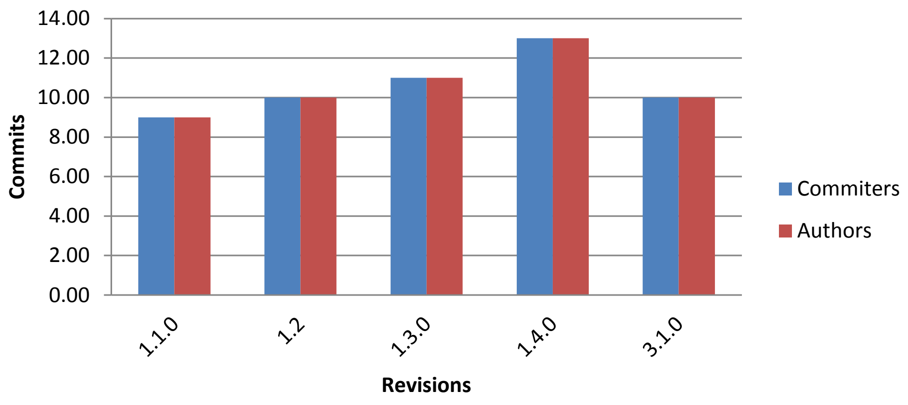

Figure B.5: Commits of commiters and authors of Solr.

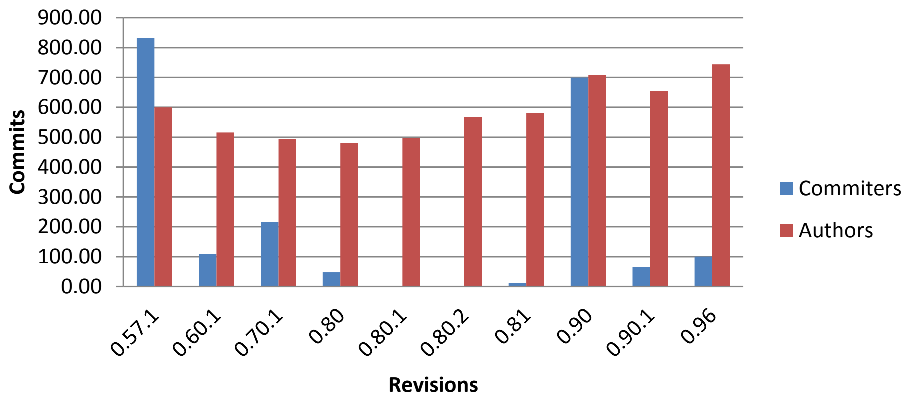

Figure B.6: Commits of commiters and authors of Voldemort.