TOOLS AND METHODS FOR LARGE SCALE EMPIRICAL SOFTWARE ENGINEERING RESEARCH

A thesis submitted to the
Athens University of Economics and Business
for the degree of Doctor of Philosophy

2012

By
Georgios I. Gousios
Department of Management Science and Technology
Athens University of Economics and Business

Contents

List of Tables

List of Figures

To my parents, for their infinite support.

Abstract

Software engineering is concerned with the study of systematic approaches towards software development and maintenance. According to many authors, software engineering is an empirical science as it strives to produce models that capture the characteristics of the development process or to predict its behaviour. Being an empirical science, software engineering is in a constant need for data.

The emergence of the Open Source Software (OSS) movement has provided software engineering researchers with rich process and product data. OSS projects make their source configuration management, mailing lists and issue tracking database systems publicly available. Although they are free to use, OSS data come with a cost for the researcher. During a lifetime spanning multiple decades, several OSS projects have amassed gigabytes of data worth studying. The computational cost for processing such large volumes of data is not trivial and lays beyond the capabilities of single workstation setups. Moreover, each project uses its own combination of the aforementioned and other project management systems management tools, such as Wikis and documentation generators. Without the appropriate abstractions, it is challenging to build tools that can process data from various projects at the same time.

In the recent years, software engineering research benefited from the availability of OSS repositories and a new stream of research that takes advantage of the rich process data residing in those repositories emerged. To evaluate the extend and use of OSS data in empirical software engineering studies, we conducted a systematic literature review. Specifically, we constructed a classification framework which we then applied on 70 randomly selected studies published in various software engineering publication outlets from 2003 onwards. The classification provided interesting insights:

- Studies are being performed almost exclusively on data originating from OSS projects.
- The vast majority of studies use data from less than 5 projects.
- There is no cross validation of the results of published works.

We attribute the obtained results to the inherent complexity of experimenting with

OSS data. To remedy the situation, we propose performing large scale software engineering research studies on an integrated platform, that combines easy to use and extend tools and readily analysed data. Drawing from the experiences of other mature empirical fields, we believe that shared research infrastructures are crucial for advancing the state of the art in research, as they enable rigourous evaluation, experiment replication, sharing tool, results and raw data and, more importantly, allow researchers to focus on their research questions instead of spending time to re-implement tools or pre-process data.

In this thesis, we investigate novel ways to integrate process and product data from various OSS repositories in an effort to build an open Software Engineering Research Platform (SERP) consisting of both software tools and shared data. We base our work on SQO-OSS, a tool designed to perform software quality analysis. We analyse the design of the raw data and metadata storage formats and as part of its implementation, we develop novel solutions to the problems of: (i) representing distributed and centralised source configuration management data in relational format (ii) identifying and resolving developer identities across data sources, and (iii) efficiently representing and distributing processing workload towards fully exploiting the available hardware.

To demonstrate the validity of our approach, and the effectiveness of the proposed platform in conducting large scale empirical research, we perform two case studies using it. The first one examines the effect of intense email discussions on the short-term evolution of a project. The hypothesis under investigation is that since OSS projects have limited human resources, intense discussions on mailing lists will have a measurable effect on the source code line intake rate. After examining the characteristics of intense communications, we construct a model to calculate the effect of discussions and implemented it as an extension to our platform. We run the study on about 70 projects and we find that there is no clear impact of intense discussions in short term evolution.

In the second case, we correlate maintainability metrics with key development process characteristics to study their effect on project maintenance. Specifically, we study whether the number of developers that have worked on a project or on a specific source module is indicative of how maintainable the project as a whole or the module is. Using the SERP platform, we run the study on 210 C and Java projects. We find no correlation between the number of developers that have worked on a project or a source code module and its maintainability, as this is measured by the maintainability index metric.

One of our findings is that in both case studies, the application of bias on the selection of the examined sample, would lead to completely different results. In fact, we show that there are more hypothesis validating cases (even though the hypotheses have been overall invalidated) for each case study than the average number of cases

evaluated per case study in currently published studies, which we derived from the systematic literature review. We consider this result as a strong indication of the value of large scale experimentation we advocate in this thesis.

Overall, our contribution has both a scientific and a practical aspect. More specifically:

- We describe a framework for classifying empirical software engineering research works and we use it to analyse the shortcomings of the current state of the art.
- We analyse the requirements and describe the design of a platform for large scale empirical software engineering studies.
- We introduce a relational schema for storing metadata from software repositories, which provides our platform with enough abstractions to retrieve software process metadata across projects and across software repositories.
- We introduce an algorithm for mapping semi-structured data from software configuration management repositories in a relational format.
- We introduce algorithms for resolving developer identities across data sources and for distributing the load of computation across nodes in a cluster environment.
- We validate our platform by conducting two case studies using it. We find that intense email discussions do not affect short term project evolution and that development team size does not affect software maintainability at the module or project level.
- We show that the results of the aforementioned case studies could be radically different if bias is applied on the selected experimentation dataset, thereby validating our thesis on the importance of conducting experiments on large scale datasets.

Finally, we make the software we developed and the data we produced available to the research community under non-restrictive licenses.

Acknowledgements

Writing a thesis is hard; when doing it under time pressure, it can be a task for the daring. Fortunately, I had been in similar situations many times in the $4\frac{1}{2}$ years I have worked at the Information Systems Technology Laboratory of the Athens University of Economics and Business, so I had in my arsenal the tools and methods to carry this task out. Here is a list of people that have contributed in more ways than they could imagine.

My "thank you" section will start with my supervisor, Diomidis Spinellis. I have been working with Diomidis since 2001, and not in a single moment did I have second thoughts. The way I see it, there are two kind of mentors: those that take you by the hand and show you the whereabouts and those that throw you in the cold sea without a life vest but provide you with the necessary support and guidance in order not to drown. Diomidis followed the second approach, and allowed me to be hit hard by paper reviewers and bureaucratic procedures, but always supported me. I believe this kind of research training made me a better researcher, in the same way hard training makes commandoes better at their job than infantry. So, thank you Diomidis for being my research boot camp trainer and for all the opportunities you gave me during all those years!

I have had other mentors too, although I doubt that they will acknowledge their role. Vassileios Karakoidas was my intro to the funded research world. Early on, he gave me one of the best pieces of advice that a noob researcher can receive: "never make another one's problem your own". Other than that, he has been a close friend, a very interesting person to discuss with and a worthy challenger on 2 player video games. The combination of his assertive quotes with the other Vassileios we had at the lab, Vlahos, faster-than-blink replies had excellent effects on morale, even though the effects on productivity remain questionable. Speaking of the other Vasilleios, he was my mentor in office, business-like, behaviour; his sense of responsibility, willingness to help and support when I was feeling blue went missing from the lab when he graduated. "Doing a PhD requires a strong gut" he used to say; how true!

I also had the chance to work with other brilliant people: Stephanos Androutsellis-Theotokis, Dimitris Mitropoulos and Kostas Stroggylos, I didn't get to know them as

well as I would like; my apologies, this is an open task for the future. Nevertheless, we had a very good time working together at the lab and even more so when going out for the occasional coffee. Panos Louridas helped me in too many occasions to remember individually with the unconceivable amounts of knoweledge he stores in his head. Eirini Kalliamvakou helped me understand that there is actually a lot of research that can be done without writing code, which in my mind initially seemed irrational. Her supervisor, Nancy Pouloudi, was kind enough to give me a personalised short "research methods for dummies" seminar, that changed the way I wrote chapter 2.

The following persons have contributed to my work without knowing it. I occasionally came in touch with them with seemingly unrelated questions, only to have their unbiased opinion: Ioannis Samoladas, Mirko Böhm and the other people from the SQO-OSS project, Israel Herraiz, Ian Rogers, Stavros Grigorakakis, Georgios Zouganelis and perhaps others that I may forget (my apologies!). Finally, I would also like to thank both organisations that supported my research.[1]

Last, but not least, Fenia Aivaloglou was my normal life counterpoise. She stood by me when I was under pressure, understood me when I told her I had to work yet another long day and provided all kinds of support from emotional to purely technical, being my resident database expert and paper reviewer. After all what we have gone through together (she also did a PhD at the same time), I believe that the difficult part finally comes to an end.

Georgios Gousios
June 2009

[1] The work presented in this thesis was supported by the Greek Secretariat of Research and Technology through the Operational Programme COMPETITIVENESS, measure 8.3.1 (Reinforcement Programme of Human Research Manpower – PENED), financed by National and Community funds (25% from the Greek Ministry of Development-General Secretariat of Research and Technology and 75% from the European Social Fund). Additional support, in the form of equipment, travel and conference funding, was provided by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software" (SQO-OSS).

Chapter 1

Introduction

empirical, adj: *based on, concerned with, or verifiable by observation or experience rather than theory or pure logic.*

— *The Oxford American Dictionary*

This dissertation is concerned with the study of tools and methods for enabling large scale software experimentation. The problem we are trying to tackle is how we enable empirical software engineering research to be performed on very large process and product datasets. What we propose is a platfrom, combining an efficient and extensible tool with preprocessed datasets, which researchers can use in order to conduct experiments with. By abstracting the raw data formats and the processes required to manipulate them, we show that complex experiments involving both process and product data can be effectively reduced to compact algorithmic descriptions of the measurements that the experiment must produce. At the same time, the processing load can be distributed to clusters of machines thereby enabling experimentation with large data volumes. The validity and applicability of our approach is demonstrated via two case studies in which we use the proposed platform as a tool for studying how process characteristics reflect on product development, on a very large data set. Our results show that large scale research with empirical data originating from both product and process data sources is possible, provided the appropriate abstractions and hardware infrastructure.

In this chapter we describe our motivation for undertaking this research, by identifying the work context and presenting our goals and contributions. We also provide a summary of each chapter, to guide the reader through the contents.

1.1 Context

The adjective "empirical" is attached to various scientific fields to denote that their primary focus is the analysis of observations made on existing systems and the introduction of models that explain the observed behaviours. While we do not refer to other sciences as empirical to signify their involvement with natural phenomena or the fact that they validate theoretical predictions by applying the scientific method, currently we do so only to characterise a specific branch of software engineering. For many years since the start of the computer age, the term software engineer was equivalent to that of a source code developer. Only when software systems became excessively complicated did software engineering emerge as a separate discipline that studies the software development process and its effects on the produced product [IEE90]. Today, similarly to other engineering fields, software engineering has taken two directions: applied software engineering is concerned with the large scale production of software while research software engineering strives to advance the state of the art by producing methods and tools that improve the software development process. The second branch of software engineering is what we refer to as empirical.

Currently, there is a growing interest in software engineering research to use data originating from OSS projects [Moc09]. It could be argued that, apart from the software landscape, the OSS movement has also changed the way the study of software is being done. OSS projects make their Software Configuration Management (SCM), mailing lists and Bug Tracking Systems (BTS) publicly available. The wealth of combined process and product data has attracted the interest of many researchers, as this is the first time in the history of software engineering that large scale empirical studies can be performed with real data outside the walls of large software organisations. However, even though they are free to use, OSS data come with a cost for researchers:

- During a lifetime spanning multiple decades, several OSS projects have amassed gigabytes of data worth studying. The computational cost for processing such large volumes of data is not trivial. For fast experiment turnover, researchers must design their analysis tools to exploit modern multiprocessor machines. However those efforts are often not enough, as the capacities of single machines are easily saturated by the processed data volumes.

- Each project uses its own combination of SCM, mailing list, BTS and other project management tools, such as Wikis and documentation systems. Without the appropriate abstractions, it is challenging to build tools that can process data from various projects at the same time.

- Empirical studies are often conducted in several phases, and a diverse set of tools

can be applied in each phase. Intermediate results must be stored and retrieved efficiently in formats suitable for use by chains of tools.

As a result of the above, several studies [kwW97, SHH+05], including ours, show that empirical studies in software engineering neither take full advantage of the data on offer nor (or at least, seldom) base their experimental results on prior studies. The same studies urge software engineering researchers to validate their models more rigorously, as this is what drives general applicability of the performed studies [PPV00]. This dissertation attempts to improve this situation by describing and evaluating a platform for software engineering research.

1.2 Contributions

It is our thesis that software engineering as a disciple should strive towards more rigorous experimentation. The OSS movement presents a unique opportunity in that respect as it provides researchers with access to rich historical process and product data originating from some of the top quality software projects in existence today [MFH02, SSAO04]. However, existing studies only partially take advantage of the existing wealth of data, which, in our view, renders the presented results vulnerable to falsification and precludes their re-use in other studies.

The main contributions of this thesis can be summarized as follows:

- We describe a framework for classifying empirical software engineering research works (Section 2.3.1) and we use it to analyse the shortcomings of the current state of the art (Section 2.3.3).
- We analyse the requirements (Section 4.1) and describe the design (Sections 4.2 — 4.3) of a platform for large scale empirical software engineering studies.
- We introduce a relational schema for storing metadata from software repositories, which provides our platform with enough abstractions to retrieve software process metadata across projects and across software repositories (Section 4.2.2).
- We introduce an algorithm for mapping semi-structured data from software configuration management repositories in a relational format (Section 4.4.1).
- We introduce algorithms for resolving developer identities across data sources and for distributing the load of computation across nodes in a cluster environment (Sections 4.4.2 — 4.4.3).
- We validate our platform by conducting two case studies using it. We find that intense email discussions do not affect short term project evolution (Section 5.1)

and that development team size does not affect software maintainability at the module or project level (Section 5.2).

- We show that the results of the aforementioned case studies could be radically different if bias is applied on the selected experimentation dataset, thereby validating our thesis on the importance of conducting experiments on large scale datasets (Section 5.3).

1.3 Thesis Outline

The remainder of this dissertation is organised as follows:

In *Chapter 2*, we describe and analyse previous work that relates to and motivates this dissertation. After a systematic literature review, we show that, despite the abundance of free software engineering data, the datasets used for conducting case studies are very small, while there is no experiment replication that would reinforce the reuse of research findings.

In *Chapter 3*, we present the problem we are trying to solve and motivate the need for a platform for large scale software engineering research. We also formulate a set of hypotheses for our model platform.

In *Chapter 4*, we present the requirements the design and bits of the implementation of the proposed platform.

In *Chapter 5*, we present two confirmatory case studies which we use as a vehicle to validate our hypotheses in two different experimental contexts. We also present evidence on the importance of conducting large scale experiments.

Finally, in *Chapter 6*, we present a list of future research topics that emerge from this work and we conclude the dissertation.

1.4 Work Performed in Collaboration

Much of the work described in this thesis has been carried out using the tools and equipment infrastructure provided by the Software Quality Observatory for Open Source Software (SQO-OSS) project, a European Commission funded research effort aiming to produce software quality evaluation tools and techniques. While the project was active, I served it as its manager, chief designer, and have contributed the largest portion of code to the SQO-OSS system's code base. However, some pieces of the work have been carried out entirely by other people while others have been carried out in co-operation with me. Specifically, I designed and developed the tool's metadata updaters, storage

data schema for metadata, plug-in architecture, project data mirroring schema, cluster architecture, and I have contributed large parts of the raw data access components. Other project participants developed various other system parts, which I have used to implement the functionality required to build the described research platform.

An overview of the SQO-OSS tool, that includes contributions made be others, can be found in Section 2.2.3.1. My specific contributions to the SQO-OSS tool that form part of the Software Engineering Research Platform (SERP) platform described in this dissertation, are analysed in Chapter 4. Additionally, the idea of applying scores to the developer identity matching algorithm presented in Section 4.4.2 came from Panagiotis Louridas.

Chapter 2

Related Work

No amount of experimentation can prove me right. A single experiment can prove me wrong.

— *Albert Einstein*

The use of the term *software engineering* can be traced back at least as far as the 1968 NATO Conference in Garmisch, West Germany. One of the pioneers in the subject, Peter Naur, defined software engineering as:

> Software engineering is the establishment and use of sound engineering principles in order to economically develop software that is reliable and works efficiently on real machines.

A more formal definition is given in the Institute of Electical and Elecronic Engineering (IEEE) Standard Glossary for Software Engineering [IEE90]:

> (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
> (2) The study of approaches as in (1).

Software engineering is mostly concerned with the study of tools and methods used to construct better software. According to several authors [Bas96, WW00, PPV00, SHH+05], software engineering is an empirical science, as both the studied artefacts and the developed methods are (or are applied on) real, existing systems. Moreover, Wohlin and Wesslen [WW00] argue, that software engineering is governed by human behaviour, as in the end it is people who develop; consequently, it is not possible to find any formal rules or laws in software engineering except when focusing on specific technical aspects. One of the founders of experiment-based software engineering, Vic Basili, notes that "software engineering is a laboratory science" and consequently he advocates a cycle of observation, model building, experimentation, and learning [BSH86].

In this chapter, we present a structured overview of the literature in the field of empirical software engineering. Near the end of the chapter (Section 2.3), we also perform a meta-analysis of the studied works to discover interesting trends and shortcomings in current empirical software engineering work.

2.1 Empirical Methods in Software Engineering

Empiricism is a branch of epistemology (a division of philosophy that deals with the theory of knowledge), whose basic assertion is that knowledge arises from experience. Empiricism emphasizes the role of experience and evidence, especially sensory perception, in the formation of ideas, while discounting the notion of innate ideas. It seeks to explore and explain natural phenomena by using evidence based on observation and logical thought. Empiricists tend to focus on the tentative and probabilistic nature of knowledge in contrast with other philosophical views of science which may be more assertive (e.g. rationalism).

Empirical observations form an fundamental part of the scientific method. The scientific method requires a researcher to observe the phenomena under investigation, formulate hypotheses, and collect data and/or design experiments to validate or falsify the hypotheses. A *hypothesis* is a reasoned proposal suggesting a possible correlation between or among a set of phenomena. A proof of a hypothesis based on empirical methods is never conclusive; however, an experiment that contradicts the predictions of a hypothesis is enough to invalidate it. An excellent study on scientific reasoning is provided by Popper [Pop35].

Empirical research approaches incorporate both qualitative and quantitative methods for analysing data; quantitative methods involve the analysis of numerical data with statistical tools in order to quantify the relationships between data groups while qualitative methods rely more on conceptual analysis of the studied artefacts. Both types of methods can, and have, been used in software engineering research. Quantitative methods are more common, as it is usually straightforward to collect data from sources related to software engineering, and set up experiments or perform exploratory studies. Quantitative research usually entails interaction with humans or design and execution of controlled experiments. Analyses of data collection methods specific to software engineering are presented by Seaman in [Sea99] and by Lethbridge et al. in [LSS05].

According to Basili [Bas96], a scientifically sound empirical software engineering study requires (1) the building of a model that explains the processes behind the studied phenomenon (2) the construction of hypotheses derived from the model and (3) the validation of the hypothesis through empirical methods. In a similar work, Perry et al. [PPV00] go in depth with what constitutes a successful empirical study; their main

contribution is that they describe in a systematic fashion the constituents of successful empirical studies in the context of software engineering. The following research methods have been proposed by several authors [WW00, GVR02, KPP+02, SDJ07, ESSD08] for empirical software engineering studies:

Experiments An experiment is an empirical inquiry that investigates relationships between variables. Specifically, experiments are characterised by the measurement of the effects of manipulating one variable on another variable and by the fact that subjects are randomly assigned to experiment actors. An experiment implies control over some of the conditions in which the study takes place and also control over the independent variables that are being tested.

Surveys A survey is the collection of information from a specific population, usually by means of questionnaires and interviews. It is useful for studying a large number of variables using a large sample size and rigorous statistical analysis. Surveys are especially well-suited for answering questions about what, how much, or how many, as well as questions about how and why.

Case Studies A case study investigates a set of phenomena in their context they have occurred. In the context of software engineering, this usually means studying the behaviour of a set of independent variables towards the construction of models that describe findings in large data sets. *Exploratory* case studies are used as initial investigations of some phenomena to derive new hypotheses and build theories, while *confirmatory* case studies are used to test existing theories and hypotheses. Confirmatory case studies are often used to falsify hypotheses, since falsification based on empirical data is more convincing than falsification based on experiment generated data. Case studies can also be characterized as *qualitative*, when they study variables that cannot be quantified, and *quantitative* otherwise.

Not all authors agree on the exact definition of each research method. For example, Glass et al. [GVR02] introduce the notion of "conceptual analysis" and appear to be using it in place of what we refer to as qualitative case study; conceptual analysis is a analytic method in philosophy that works out a problem by decomposing it into its constituent parts and identifying their relationships. Similarly, a qualitative case study will try to identify relationships between non-quantitative characteristics by breaking them down to their constituents. Runeson and Höst [RH09] classify experiments as special cases of case studies; in our opinion, quantitative case studies differ from experiments as the data a case study is performed upon already exist, while in the case of experiments, the data required to validate a hypothesis do not exist and the experiment is conducted in order to generate them.

Table 2.1: Empirical research methods as used in software engineering works.

Study	Sample Size	Empirical	Experiments	Surveys	Case Studies
[kwW97]	> 600	—	3%	—	10.3%
[GVR02]	369	14%	3%	1.6%	2.2%
[SHH+05]	5453	15%	1.9%	$< 1.1\%$	12%

Furthermore, Easterbrook et al. [ESSD08] describe *ethnography* and *action research* as other potential research methods; in agreement with [RH09], we believe that ethnographic studies is a particular approach to case study research that focus on development teams or communities practices. We also fail to see how action research differs as a research method from case studies in the context of software engineering. Action research requires the researcher to manipulate the behaviour of participants under investigation through his own participation towards improving it while the study is running. This implies that the observed phenomena falsify the model under investigation in some manner. The continuous model evolution approach taken by action research is conceptually a repetitive confirmatory case study, with a new investigation subject (the refined model) in each iteration.

As a consequence of the definition babel, it is not easy to compare the findings of exploratory literature studies. Three surveys on the use of empirical research methods have been carried out by Sjøberg et al. [SHH+05], Glass et al. [GVR02], Zelwowitz and Wallace [kwW97]. The findings are summarized in Table 2.1. From of a total of 5453 scientific articles published in 12 major software engineering journals and conferences in the decade 1993–2002, Sjøberg et al. identified 113 controlled experiments, reported in 1.9% of the study corpus, in which humans performed software engineering tasks. Glass et al. investigated 369 works and found only 3% of them reporting on controlled experiments. Zelkowitz and Wallace [kwW97] found similar numbers by examining more than 600 papers. The use of surveys as a research tool is equally minimal; only 1.6% of papers are reported as survey by Glass et al. and less than 1% by Sjøberg et al. Furthermore, case studies occupy 2.2%, 10.3% and 12% of the total number of research papers respectively. From the reported findings, it becomes apparent that the overall use of empirical methods in software engineering is very small for a science field that is primarily using empirical data.

2.2 Ingredients of Empirical Studies

In this section, we describe in a structured format the constituents of empirical studies. From reading and analysing over 200 works (a selected sample of which is presented in Table 2.8), we observed that most empirical studies, especially case studies and empirical validations of models and tools, follow a certain recipe:

Study = *Model* + Metrics + Data + Tools + Analysis approaches + *Results analysis*

Researchers working with empirical data have access to a limited set of data sources, tools and data extraction paradigms. The important outcome of each empirical research is usually not a new data source or a new data extraction method, but insight from applying and validating a newly discovered model to existing sets of data, thereby discovering new knowledge. This observation is intuitive, but sufficient to organise the presentation of the related work around it. In the next sections, we present a structured overview of the ingredients common to all empirical studies.

2.2.1 Metrics

Metrics are measures of software properties or its specifications. There are many characteristics of software and software projects that can be measured, such as size, complexity, quality and adherence to process [Kan03, Lai06]. What to measure is a common question in all empirical studies; the Goal Question Metric (GQM) approach [BCR94] has been widely used to assess a metric's suitability towards answering a research question. In the following sections, we describe the various software metrics that are used in empirical software studies. Metrics are usually divided in three broad categories [FP98, Kan03]:

Product metrics quantify various characteristics of the software. They are divided in two categories [FP98]:

- Internal attribute metrics, that access the size and structure of the product.
- External attributes metrics measure how the internal attributes of the software reflect to its quality.

Process metrics are metrics that refer to the software development activities and processes. Measuring defects per testing hour, time, number of people, etc. falls under this category.

Resources [FP98] or **Project** [Kan03] metrics are metrics that refer to any input to the development process (e.g. people and methods).

In the following sections, we present the most commonly used metrics in empirical software engineering studies.

2.2.1.1 Product Metrics

Product metrics measure internal attributes of the software under investigation. Product metrics do not reveal any important information if reported standalone; for this reason

most empirical studies use product metrics to assess the effectiveness of development process modeling or tuning. The two most important internal properties to assess are size and structure.

Internal Characteristics - Size Size is a fundamental attribute of software. Size measurements are important to compare project sizes, evaluate the amount of effort that was required to develop the software or evaluate team productivity. The size of a project is also usually correlated with defect counts, security flaws and evolutionary metrics. Several metrics of software size have been proposed, which measure program words [SEG68], lines or functionality.

The simplest way to measure software size is by summing up the Lines of Code (LOC) of the source files that comprise the software. There are two major types of LOC measures: physical LOC and logical LOC. Physical LOC is the total number of lines in a source module, while logical LOC is usually measured as the number of program statements. The LOC metric is overly simplistic as it does not account for comments, licence notes or code formating and, more importantly, in expressive differences between the various programming languages. To account for those problems, researchers have produced frameworks for counting lines of code in an integrated manner [Par92] and devised length equivalence tables between programming languages [Jon91]. Despite criticism, the LOC metric is widely used in empirical studies, especially so in software evolution research.

Halstead in his classic "Elements of software science" [Hal77] work extended the notion of software size to include other code elements such as source code operators and operands. For Halstead, the length (N) of a program is equal to the total number of operators (N_1) and operands (N_2), while the program's volume (Halstead Volume (HV)) is equal to $N \times \log_2(\mu_1 + \mu_2)$ where μ_1 and μ_2 is the number of *distinct* operators and operands respectively. The program volume is essentially a measurement of the length of an algorithm implementation without being based on code layout. Since their inception, Halstead's metrics have been a constant target for criticism in the literature [FL78, Wey88, CALO94, FP98, MSCC04, Her08], but they are popular in the evaluation of software maintainability.

The previous size measures count physical size: lines, operators and operands. Many researchers [AG83, Cha95] argue that this kind of measurement can be misleading, since it does not capture the notion of how the counted physical entities reflect on the function the software carries out. Albrecht [Alb79] developed a methodology to estimate the amount of functionality that is performed by software, in terms of the data it uses and generates. The function is quantified as function points, i.e. as a weighted sum of the numbers of inputs, outputs, master files, and inquiries provided to, or generated by, the software. Function point analysis has been used for planning

and for productivity assessment, although it has been criticised as difficult to apply and automate.

Internal Characteristics - Structural Complexity One of the first and most widely accepted metrics for software complexity is the McCabe Cyclomatic Complexity (MCC) [McC76]. The metric examines the control flow graph of a function and calculates a measurement of its complexity by enumerating the number of possible execution paths a program function has. The Extended McCabe Cyclomatic Complexity (EMCC) metric also considers the number of boolean operations in the total number of execution paths.

Henry and Kafura's Information Flow (IF) [HK81] metric relates a module's complexity to the number of cross-references between the module and other modules. Specifically, given the number of modules that call functions in the given module (f_i) and the number of modules that are used by the given module (f_o), it defines the IF for the modules as $IF = (f_i \times f_o)^2$. The IF metric is essentially a measurement of a package's or program's (depending on the calculation scope) structural complexity.

Based on various approaches to measure the structural and modular complexity of software, Card and Glass [CG90] proposed an integrated system complexity model. For Card and Glass, system complexity is the sum of structural and data complexity. Structural complexity is defined as the mean fan-out factor for all system modules and data complexity is proportionate to the number of I/O variables and inversely proportionate to the fan-out of a module. They also derived a function that correlates the results of their model with the project's error rate.

All structural metrics discussed so far were initially targeted to the assessment of procedural programs. With the rise of object-oriented programming, software metrics researchers tried to figure out how to measure the complexity of such applications. The most widely known set of metrics for object-oriented programs is the Chidamber and Kemerer (CK) metrics suite [CK94]. The metrics that CK propose are:

Weighted Methods per Class (WMC) The sum cyclomatic complexity for all methods visible in a class. As method visibility is difficult to assess before runtime in many environments, for example due to dynamic loading, the WMC metric is usually set equal to the number of methods in a class definition without considering the super classes.

Depth of Inheritance Tree (DIT) The length of the maximum path of class hierarchy up to the evaluated class.

Number of Children (NOC) The number of immediate subclasses of a class.

Coupling Between Objects (CBO) The number of classes a class depends upon ("imports" or "includes" in Java and C++ terminology respectively).

Response For Class (RFC) The total number of different methods that are being called from public method bodies in response to a message being received by an instance of a class.

Lack of Cohesion in Methods (LCOM) The number of different methods within a class that reference a given instance variable.

Even though numerous toher object oriented have been proposed in the literature [FP98], the CK metrics suite is by far the most popular for assessing the design quality of object-oriented software. Several studies show that the CK metrics suite assists in measuring and predicting the maintainability of object oriented systems [BBM96, BMB96, FP98, Spi06b]. Moreover, Rosenberg et al. [RSG99] present a set of thresholds for each metric; if a class scores worse than the threshold in any 2 metrics, it should be a target for refactoring. Finally, studies show that certain CK metrics are linked to faulty classes and can help predict such [Kan03].

External Characteristics - Quality Quality is a functional and aesthetic measurement, used, for instance, to specify a user's satisfaction with a product, or how well the product performs compared to similar products. Quality is a very difficult property to measure because humans tend to understand quality as the level of the productb••s conformance to their expectations. Software quality is formally defined by the ISO/IEC 9126 standard [ISO04] as comprising of six high level characteristics, namely functionality, reliability, usability, performance, maintainability and portability. The standard also defines a set of attributes for each characteristic. Not all ISO 9126 characteristics are of concern in software engineering, mainly because they cannot be quantified statically (e.g. performance) or at all (e.g. functionality). Below, we refer to the two characteristics that have received the most attention in software metrics bibliography [FP98, Kan03], namely maintainability and reliability. A more thorough examination of the field of software quality with respect to the ISO 9126 standard is provided by [Spi06b].

Maintainability Maintenance is the software life cycle stage that begins after the software has been successfully released. It is believed to occupy the majority of the software's life cycle [Hat98], thereby incuring most of the development costs.The processes that take place during software maintenance are officially described in [**?**]. The IEEE glossary for software engineering [IEE90] defines 3 types of maintenance operations: corrective, perfective and adaptive. Based on these definitions, Coleman et al. [CALO94] define maintainability as the ease at which software can be modified in order to fix and remove defects, adapt to a new operating environment, meet new

requirements, and improve the overall software structure in order to make the future maintenance easier (refactoring [Fow99]).

Coleman et al. [CALO94] provide a widely known formula for calculating a measure of maintainability for a given software system, the MI:

$$\begin{aligned} MI &= 171 - 5,2 \times \ln(avg(HV)) \\ &\quad -0,23 \times avg(MCB) \\ &\quad -16.2 \times \ln(avg(LOC)) \\ &\quad +50 \times \sin(\sqrt{2.4PerCM}) \end{aligned}$$

$avg(HV)$ is the average Halstead Volume per module, $avg(MCB)$ is the average extended cyclomatic complexity per module, and $avg(LOC)$ is the average lines of code per module. The $PerCM$ metric denotes the percentage of comment lines in the total number of lines in the project.

The MI is a composite metric whose constants were derived through experimentation with large C/C++ data sets. The validity of the individual metrics that compose the maintainability index has been an issue of scrutiny [SI94, Wey88, FL78], and therefore their composition is dubious. Each base metric has been particularity selected to account for the inability of the rest to measure software growth factors: for example the MCB metric's, insensitivity to program length is counterpoised by HV inability to capture program structure. The maintainability index is regarded as non-suitable for object oriented languages, since various stydies have shown that object oriented structural metrics are better for assessing maintainability [BBM96, DJ03].

Reliability Software reliability can defined as the probability of failure-free software operation for a specified period of time in a specified environment. Reliability is a widely studied external characteristic, as it is related with one of the most important properties of software: correctness. According to [Kee94], software reliability is very difficult to assess preemptively (as is the case with hardware reliability) and impossible to reinforce by means of redundancy, since software as a product depends mostly on human factors rather than natural laws and pre-existing components.

A simple reliability metric is the defect density; it is expressed as the number of defects found per certain amount of software. This amount is usually counted as the number of lines of code of the delivered product. Many researchers split the kind of defects into two categories: known defects, which are the defects that have been discovered during testing (before the release of the product) and latent defects, which

are the defects discovered after the release of product. For each of these two categories, there is a separate defect density metric. More complex reliability models are described in references [FP98] and [Pel01]. Most models attempt to model reliability as a function of basic product measures such as size and/or complexity.

2.2.1.2 Process Metrics

Defect Removal Effectiveness (DRE) denotes the development team's ability to remove defects. The metric is defined as: $\frac{N}{N+S}$ where N is the number of defects found during the development of the software while S is the number of errors found after end of the software development cycle. The granularity at which N and S is calculated depends on the development process being used; for example, in a waterfall-based process, S would be calculated after the delivery of the software. In more iterative development cycles, S would be calculated after each iteration cycle. The Backlog Management Index (BMI) is a measure of the effectiveness of a development process to deal with incoming defects. The BMI is defined as the ratio of the number of problems closed to the number of problems arrived in a specified time frame. Closely related is the Mean Time To Repear (MTTR) metric, which measures the average time to resolve an issue. Obviously, higher values in all metrics denote both an effective development team and/or software which is easy to maintain.

2.2.2 Data

Empirical studies use a variety of data sources, which fall under one of the following categories:

Product data is the direct outcome of the development process. It mainly contains the source code but can also include other artifacts such as image files, tool configuration files, documentation files etc.

Process data is a by-product of the development process. Examples include data maintained from tools developers use to manage the project versions (SCM systems), electronic communication trails (mailing lists, Instant Relay Chat (IRC) logs), organisational memory records (Wiki systems, document archives) and BTS.

Not all kinds of data enjoy equal use in empirical studies. Naturally, product data have been available for study since the beginning of the software engineering discipline, so empirical studies started as early as 1960. Since the wake of the OSS movement from 1990 and onwards, a lot of software is being written using tools and methods originating in distributed development environments. Moreover, a vast amount of OSS data have been made available on the Internet, which makes their use suitable for researchers. A

new wave in software engineering research, colloquially referred to as Mining Software Repositories (MSR), explores the use of process data originating from software tools in conjunction with product data to evaluate the development process and the quality of software projects. As we show in Section 2.3, most empirical studies use software repositories to extract facts about the software development process and to create models of software evolution.

2.2.2.1 Source Code

Source code is the main outcome of the development process. As such, it is the target for the application of product metrics described in Section 2.2.1 and clone detection methods described in Section 2.2.4.4. The source code of a project represents a snapshot of its development lifeline, which in itself is not a very rich source of information for empirical studies. By applying tools on source code, one can obtain information about the size, the complexity and the structure of the project. Further analysis of source code artefacts can yield authorship information at the file [FSGK06] or line [GPGA09] level, the license under which it is distributed [Sca04], the size and coverage of code documentation, or the existence of "code smells" [Fow99, MVL03] (pieces of code that need refactoring).

2.2.2.2 Source Code Management Systems

SCM systems store a project's source code along with meta information about the development process.

The typical workflow when working with an SCM system is the following: The *checkout* command retrieves the latest version from the project's repository and creates a workspace. The *update* command updates the current workspace state: it retrieves all changes since the last checkout or update and creates new files, replaces outdated files, and removes files that have been deleted in the repository. If a file has been changed in both the workspace and the repository, modern SCM systems try to integrate the changes automatically at the line level; a *conflict* can occur when a file has been concurrently modified at the repository and on the developer's workspace. The *commit* command submits changes made by a developer to the repository. *Changesets* can modify, add or remove files; in more advanced SCM systems (e.g. Subversion (SVN)) changesets also include file copies between repository locations. A *tag* is a pointer to a particular state of the project under revision control, for example it can be added to signify a released version of the project. Finally, when developers want to test new features that might render the source code base unstable, they usually create a *branch*, which is essentially a fork of the code base at the moment the branch is created. When the new feature has been tested, the branch is *merged* to the main code base tree.

SCM systems can be divided in three categories, based on the type of access they provide developers with:

Centralised systems keep all versions and metadata in a central location (usually an internet server) and permit concurrent access of multiple developers to the repository contents. The repository acts as the central depot of project artefacts and consequently maintains a centralized view of project history. First generation centralised SCM systems maintain state per file and therefore it was not possible to retrieve the state of the whole code base after a commit; newer systems maintain state in a project wide fashion. Examples include, the Source Code Control System (SCCS) [Roc75], the forefather of all SCM systems, the Concurrent Version System (CVS) [Fog99] and the SVN [PCSF08] system.

Distributed systems combine a local SCM with remoting functionality that enables developers to manage versions locally but also to share their changes with others online. Distributed SCM systems do not have the notion of project history; in fact, a project might have multiple parallel lifelines that only materialise if they are published and then merged with the main development tree. As of this writing distributed SCM systems are very new and have not yet been used widely for research purposes [BRB+09, Moc09].

In typical analysis scenarios, software repositories are not used directly; it is usually more efficient to extract semi-structured information from the SCM repository and store it in a relational or raw text format [FPG03, ZW04, RKGB04, Ger04b, JKP+05, Moc09, **?**]. The typical method of converting project repository data to relational formats involves the following steps:

- Extracting the revision log from the repository. The revision log contains information about the files that have been changed by each commit along with meta information about the change, such as the author or the time of change.
- Identifying project revisions: The revision log is used to extract information about the currently processed revision, such as whether the revision includes branching or tagging operations.
- Recreating file state: Using the revision log and pre-processed data from previous revisions, the processing algorithms recreate the project's file tree at any instance in its lifetime.
- Resolving transactions and revision order: First generation centralised SCM systems do not have the notion of project state, so it is necessary to derive it in the form of a transaction for each commit [ZW04, Ger04b]. On the other hand,

distributed SCM systems do not have the notion of project lifetime; in that case an approximation based on commit timelines must be implemented.

SCM systems are goldmines of information, containing both process and project data. For the point of view of researchers, it is like a time-machine that they can use to travel back in time to examine the state of the project at specific points in time and correlate these states with process metrics. SCM system data have been used to evaluate maintainability [MV00, NM03, PP05] and refactoring operations [KWB05], to perform software archaeology studies [RGBH05] and to extract developer turnover information [ARGB06, **?**].

2.2.2.3 Electronic Communication Trails

Software developers use electronic communication systems in order to increase immediacy in information exchange. This applies especially in global software development projects, where time and distance can be significant hurdles in project development [Spi06a]. The electronic communication tools developers most often use are electronic mail, IRC for group discussions, and Instant Messaging (IM) for person to person discussions. Discussions on IRC and IM are considered informal and personal and thus projects seldom maintain discussion logs; consequently, there has been a limited number of studies due to the lack of data [SJH09]. On the other hand, email message exchanges are usually archived and in the case of OSS, those archives are usually available to browse and download. Public mailing list data have been used in a number of empirical studies either standalone [LM03, VTG+06, SSA06, WM07], or combined with data from other data sources [CH05, WND08].

Mailing list data is neither easy to obtain nor to process. A variety of mailing list management software packages is in use by OSS projects while archives are usually published through a custom web page per project. There are no programmatic interfaces that can be used to automate the process of retrieving mailing lists. Additionally, the few web sites that collect emails from OSS projects, like the Mail Archive (MARC)[1] and Nabble[2], only allow browsing through the archives on a per message basis. In general, retrieving email archives requires a custom process for each project, which results to a prohibitive cost for using mailing list data in large scale empirical research. Moreover, the emails that comprise the archives do not always follow the standard [Cro82] for the message header contents, while extracting semantical information from unstructured text in the message body requires natural language processing and information retrieval techniques to be employed. For those reasons, emails are primarily used for studying the social aspects of software development instead of the development process itself,

[1] `http://marc.info`
[2] `http://www.nabble.com/`

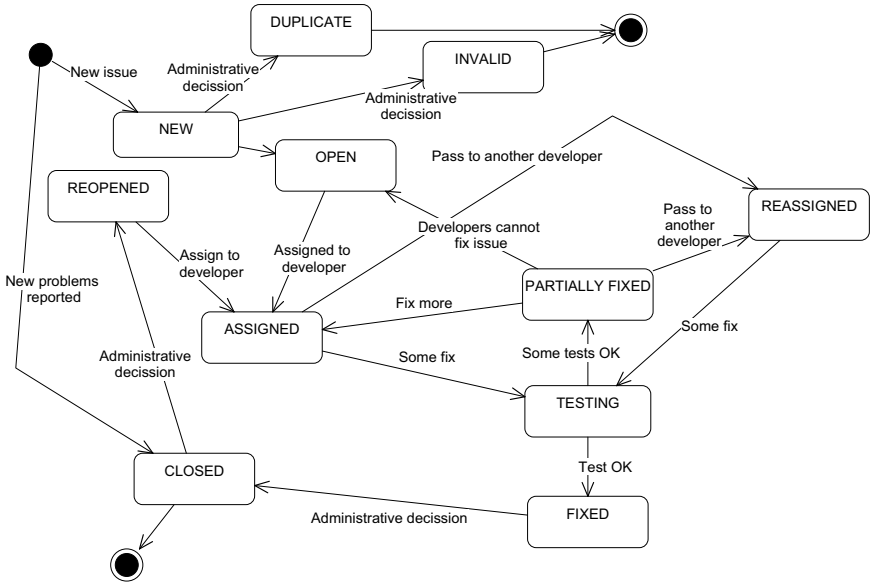

Figure 2.1: Possible bug states in BTS systems

which only requires the extraction of thread participation relationships from the mail headers.

2.2.2.4 Bug Tracking Systems

BTS are used to collect bug reports and feature requests from users and developers in a manual or semi-automatic manner (e.g. through application crash handlers). Most BTS are organised around bugs reports; each new bug is given a unique identification number and this is what developers use in order to refer to it. As shown in Figure 2.1, BTS systems maintain a set of states for the bug reports they receive. Apart from state, a bug is usually assigned a fix priority and a severity level. Users and developers are also allowed to comment on the bug's resolution progress. Most popular BTS systems (currently Bugzilla, Trac and others) maintain a database with all bug related activity and feature user and programmatic interfaces to access and query data in the database.

BTS are a very important source of information for empirical research as they contain both product and process data in a structured format that can be easily processed. Product data is the actual defects in the project itself or in its subsystems; process data can be extracted by analysing the characteristics of the defect fixing operation, for example the time required to fix critical defects or the number of bugs that are open at a given moment. Records in BTS can be correlated to entries in SCM systems by analysing commit messages [MFH02, GM03, SZZ05] or by analysing the contents of bug reports and finding the files they affect [CC05].

2.2.2.5 Pre-processed Data Repositories

A recent development in the field of data sources is the provision of research-oriented pre-processed datasets. As empirical research has become more sophisticated and researchers often need to explore properties of large datasets originating from OSS projects, experiments are becoming increasingly difficult to setup. Collecting and re-processing data, calculating metrics, and synthesizing composite results from a large corpus of project artifacts is a tedious and error prone task lacking direct scientific value. In other fields of computer science, researchers use predefined data sets (e.g. the Association for Computing Machinery (ACM) Knoweledge Discovery in Databases (KDD) cup datasets for data mining research [ACM]) or software platforms for developing and executing experiments. Both pre-configured datasets and research platforms lead to experiments that are easy to replicate and also to research results that are comparable and that can be extended by further research. Two research projects are currently working on providing canned datasets for facilitating research, namely FlossMole and FlossMetrics.

The FlossMole [HCC06] project was first to provide an open research data set. It collects and processes data from several OSS software forges (SourceForge, ObjectWeb and the Free Software Foundation (FSF) among others) and consolidates them in an integrated database. As the project does not have internal access to the forges, the datasets it provides are a mirror of what can be accessed through each forge's web page. Nevertheless, the data is useful for research not involving access to source code artefacts.

In a similar fashion, the FlossMetrics project [GSy] provides data from several OSS projects. The FlossMetrics database is derived by downloading the SCM repository, the full mailing list archives and the bug database for each project and running the CVSAnaly [RKGB04], the MlStats and other tools on each project resource. The resulting databases are offered for downloading. The databases contain structured views from unstructured repository data and simple source code metrics, such as cyclomatic complexity and Halstead's Volume. The FlossMetrics datasets offer a good source of research data, even though the databases are separate for each project and thus do not readily enable research across projects.

2.2.3 Metric Tools and Measurement Automation

Metrics calculation is a prerequisite to almost any study involving empirical data. The tools available to calculate metrics vary from language specific to language independent and from standalone to integrated into development environments. The majority of metrics tools calculate structure and size metrics and work for languages such as C++ and Java. A common characteristic of the majority of the available tools is that they

only calculate metrics on source code checkouts rather than SCM systems and thus they require extra effort to link the measurements to specific project states. A notable exception is the Columbus framework [FSG04] that calculates over 80 C++ metrics and is extensible to other languages.

Researchers working with empirical data understood early on that standalone product data measurements would not suffice. A variety of tools have been developed to automate the process of extracting and processing data from SCM systems. The CVSAnaly tool by Robles et al. [RKGB04], converts information from CVS repositories to a relational format. CVSAnaly works in three steps; it first parses the CVS log, then it cleans the data and extracts semantic information from it, and finally it produces statistical data about the project. During the first step, CVSAnaly extracts semi-structured information such as the files affected by each commit along with characteristics of the commit, for example whether it applies a patch or resolves a bug using text-based heuristics. During the semantic reconstruction step, CVSAnaly infers information about commit transactions using the sliding window algorithm [Ger04b, ZW04] and resolves duplicate developer names. Finally, it produces a set of simple statistics to reveal interesting facts about the project's history. CVSAnaly has been used to study properties and characteristics of the OSS development process in several studies [Mas05, HRA+06, ARGB06, RGBM06].

The Hackystat tool [JKP+05] was the first effort that considered both process and product metrics in its evaluation process. Hackystat is based upon a push model for retrieving data, as it requires tools (sensors) to be installed at the developerb••s site. The sensors monitor the developers use of tools and updates a centralized server. As Hackystat was designed as a progress monitoring console rather than a dedicated analysis tool, it cannot be used for post-mortem analyses, like the ones performed with OSS software. However, it provides valuable information about the software process while it is developed.

The Release History DataBase (RHDB) [FPG03] was first to combine data from more than one data sources, namely from bug databases and SCM systems. Similarly to CVSAnaly, the RHDB tool uses the CVS log to extract and store to a database information about the files and versions that changed. It improves over CVSAnaly in that it stores information in a format that allows it to recreate the project state for each project version on-the-fly. The process of combining the data is relatively simple: every time a reference to a bug report is discovered (through heuristics) in the revision log, the system retrieves and processes the bug report and stores a link to the affected file. A similar tool is SoftChange [GM03, Ger04b, GH05]. Softchange extracts data from an additional datasource (Changelog files) and infers facts from the source code once the data extraction is finalised.

Table 2.2: Tools used in empirical software engineering research

	Data Sources				
Tool	SCM	Bug	Mails	Other	Num Projects
CVSAnaly	✓				1
Hackystat	✓				1
RHDB	✓	✓			1
Softchange	✓	✓		ChangeLog	1
Hipikat	✓	✓	✓	Docs	1
Kenyon	✓				>1
Map Reduce	✓				>1

The Hipikat tool [CM03, CMSB05] was designed to act as an automated store of project memory. It provides artifact-based search for project-related artefacts. It combines structural relationships with relationships found by a measure of textual similarity. To build the search index, the Hipikat server imports data from source code repositories, mailing lists, bug management databases and project documentation and extracts and persists links between all project artefacts. The Hipikat tool can then be queried to return all artefacts that are related to a specific artefact that the user is currently working with. A similar recommendation system was built by Microsoft [Ven06] using internal tools and data sources, although no further details have been made available.

A notable tool, architecturally similar to the one described in this thesis, is Kenyon. From the limited documentation available [BWKG05, Bev06], it appears that Kenyon is a platform that pre-processes data from various types of SCM repositories, in a unified schema and then exports the database to other tools which are invoked automatically. The Kenyon database is specifically tuned for studying source code instability factors.

Finally, in recent work [SJAH09], Shang et al. attempted to combine large scale, distributed processing paradigms, such as Map-Reduce [DG04], with standalone tools in order to facilitate research with large data volumes. They evaluated this approach by porting an available tool to the Hadoop framework, an implementation of the Map-Reduce paradigm. They find good experiment execution speedup, but this approach requires modifications to existing tools, careful selection of mapping and reducing strategies through experimentation, and may not be suitable for analysis algorithms where the analysis result of a single state depends one the results of the previous state.

Table 2.2 presents a comparative evaluation of the tools that have been used for empirical studies. Most tools are used to analyze product and historical data originating from SCM repositories.

2.2.3.1 The SQO-OSS Tool

In this section, we describe the SQO-OSS tool, an extensible tool targeted to software quality evaluation. We used SQO-OSS as the basis of our work and moreover we implemented large parts of its functionality. Our original contributions to the tool and their scientific value are described in detail in Chapter 4; here, we provide an overview of the tool and briefly go through its design.

SQO-OSS is a tool that incorporates data from SCM repositories, mailing lists and BTS databases. It was specifically designed and implemented to support product and process data analysis on large datasets. It is also extensible through plug-ins that calculate metrics by combining metadata and raw data from all three datasources or from other metrics. It also automates the plug-in excecution and can distribute the processing load on clusters of machines. In comparison to the tools presented above, SQO-OSS is more advanced in almost every respect, ranging from the data volumes it can process, to the fact that it can fully automate quality evaluations and also supports interfaces for controling and viewing the results of the analysis tool executions.

The tool's architecture can be seen in Figure 1. To separate the concerns of mirroring and storing the raw data and processing and presenting the results, the system is designed around a three-tier architecture:

Tier 1 is the data layer. It contains software components that organise raw data and manage metadata and results. Different data storage formats are used for raw project data and metadata. They are described in detail in Section 4.2.

Tier 2 is the processing layer. It contains the necessary components to support the execution of custom analysis tools (from thereon: plug-ins) and provides abstractions to the underlying data.

Tier 3 is the results and metadata presentation layer. It allows external clients to connect to the system in order to retrieve metadata and analysis results.

The core's role is to provide the services required by metric plug-ins to operate. The core is based on a service oriented architecture; various services are attached to a component based infrastucture and are accessed via a service interface. Implementations of services are fully independent from the service interface and in fact implementations can be removed or altered at runtime while only affecting the service clients. The core can also host various versions of service interfaces at the same time to cater for scenarios where minor updates in service interfaces affect the operation of plug-ins. The core provides the runtime platform for plug-ins. Plug-ins are loosely coupled with the core; they are discovered and enabled during startup but they can be disabled at any time

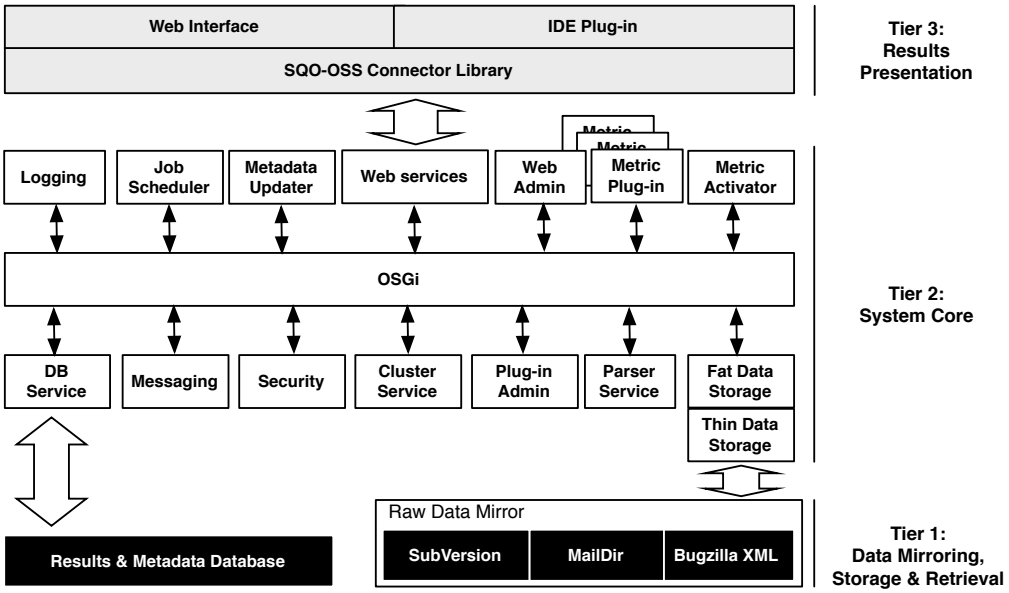

Figure 2.2: The SQO-OSS system architecture

while the system is running. The core implementation is based on the OSGi [OSG07] component model.

The SQO-OSS tool implements the following services:

Data Access Stack The data access stack consists of two basic components: the database service and the fat/thin data access stack. Access to raw data is regulated by a plug-in based stack of accessors whose lower layers directly touch the mirrored data (Thin Data Store—TDS), while the higher parts provide caching and combined raw data and processed metadata access (Fat Data Store—FDS). The TDS service abstracts the data offered by the raw data stores in an intermediate format and provides an object oriented interface to both the data and the data stores. It is based on accessors, libraries that know how to access the underlying data formats, and supports extensions through plug-ins. On the other hand, the FDS service provides a data-agnostic, higher level service: it applies on metadata formatting filters that return structures representing the semantic relationships of the data. For example, the `InMemoryCheckout` formatter would create an in-memory tree representation of the project's files that would be equivalent to retrieving it from an on disk checkout of the same version of the project. The FDS is a very versatile and powerful tool as it creates structures using metadata in the database, while it can also filter out unwanted entries.

The database service is central to the system as it serves the triple role of abstracting the underlying data formats by storing the corresponding metadata, storing metric

Table 2.3: Implemented updaters in SQO-OSS

ID	Updater	Input Data	Affected Tables	Phase	Depends
1	SVN	An SVN repository instance	`ProjectVersion`, `ProjectFile`, `Developer`	1	—
2	Maildir	Multipurpose Internet Mail Extensions (MIME) mail messages stored in a `maildir`-formatted directory	`MailMessage`, `Developer`	1	—
3	Bugzilla reports	A directory containing eXtensible Markcup Language (XML) formatted bug reports, one file per report	`Bug`, `BugSeverity`, `BugPriority`, `BugStatus`, `BugReportMessage`, `Developer`	1	—
4	Ohloh Devs	`ohloh.net` data, in XML format	`OhlohDeveloper`	1	—
5	Developer Matcher	The `Developer` and `OhlohDeveloper` table	`Developer`, `DeveloperAlias`	2	1,2,3
6	Mailing Thread	The `MailMessage` table	`MailingListThread`	2	2

results, and providing the types used throughout the system to model project resources. It uses an Object Relational Mapping (ORM) to eliminate the barrier between runtime types and stored data [O'N08], and has integrated transaction management facilities. ORM facilitates plug-in implementation by transparently converting simple queries to method calls and hiding important queries through method implementations. It also enables navigation by means of method calls among entries in the object graph that feature parent-child relationships.

Metadata Updater The SQO-OSS system uses both raw data from the projects it measures, and metadata derived from the raw data. Metadata must be updated every time the raw data are updated. The job of the metadata updater service is to maintain consistency between the raw data and the metadata. The metadata update service is one of the few services that receive external input. It does so in order to get notified by external tools when raw data updates have finished. The processing of the update request is performed by a component that validates input and schedules the appropriate update job depending on the request and underlying data format.

The SQO-OSS tool hosts two sets of metadata: those that map directly to raw data and those that are the result of extracting semantic relationships from the metadata. Consequently, the metadata update is a two phase process. In the first phase, metadata are extracted from the raw data and are stored in the system database. Second phase

updaters work on the imported metadata and analyse them in order to extract information that is hidden by the one-to-one mapping approach that is used by phase one updaters. Examples of second phase updaters include the mailing list thread updater and the developer matcher. As a result, a large number of MSR tasks described in Section 2.2.4.1 can be implemented as a second phase updater in a cross-project manner in SQO-OSS, except if the analysis method uses tool-specific data. Table 2.3 lists all currently implemented updaters.

Job Scheduler One of the most important functions SQO-OSS performs is the splitting of the processing load on multiple CPUs. Typically, all plug-in invocations and a considerable number of maintenance tasks are run in parallel. A job scheduler component is required to manage the sharing of processors among tasks as the rate of task generation can at any time be higher than the rate of task consumption, in which case a direct task-CPU assignment using operating system provided semantics would overwhelm the operating system.

All tasks in SQO-OSS are modelled as jobs. A *job* is a generic abstraction of an executable entity. Each job maintains a list of dependencies on other jobs and is assigned a priority level. The scheduler uses this information to order the execution of jobs. A job can be in a finite number of states, which can only be modified by scheduling decisions. Other SQO-OSS subsystems can implement custom jobs by providing implementations of the task to be executed and assigning those jobs to the scheduler for execution.

The job scheduler service maintains a task queue and a configurable size worker pool and assigns tasks to idle workers. The scheduler is neither required to implement task fairness policies nor to pre-empt long running tasks. For this reason, it does not maintain information about the runtime devoted to its task. It does however need to resolve which job should execute next in an efficient manner. The scheduler maintains two job queues, the work queue, which is a priority queue data structure whose head always contains the next job to execute and the blocked queue that contains jobs that have been blocked due to unsatisfied dependencies. By default, when a job is enqueued, its dependencies are checked: if it does not have any dependencies or its dependencies have already finished executing, then the job is added to the work queue directly, otherwise it is added to the blocked queue. Upon successful job execution, the scheduler informs the jobs that depend upon it about the event and the jobs must check their dependencies to decide whether they can be executed or not. The scheduler will use this information to move the job from the blocked queue to the work queue. This type of asynchronous scheduling decisions effectively substitutes classic dependency resolution algorithms (for example, topological scans) on large work queues while also allowing the execution of scheduling decisions on multiple CPUs (the CPU that finished executing

a job also performs the dependency resolution steps), with minimal synchronisation.

A more thorough, architecture-level description of the job scheduler is provided in [**?**, Chapter 12].

Auxiliary Services The SQO-OSS tool features several auxiliary services. The logging service offers a customisable, centralised diagnostic output component. The plug-in administration service manages plug-in configuration and executes the plug-in registration routines on behalf of the plug-in. The metric activator service is responsible for resolving metric plug-in dependencies and scheduling metric jobs with the appropriate order on user request or when a metadata update has finished. The messaging service sends short messages (for example emails) using configurable backends.

Interfaces The SQO-OSS tool has two kinds of interfaces, for controling the evaluation process and for presenting the evaluation results. The first type of service is provided by both a user accessible interface and a programmatic one that is used for scripting and automating the evaluation process. The results presentation interface is provided by a dedicated component over an XML-RPC (web services) transfer. The web services interface is used by the upper layers to render the metric run results. Currently, clients include a web site and an Eclipse plug-in.

2.2.4 Analysis Approaches

2.2.4.1 Repository Mining

Even though the name might suggest so, SCM repository mining is not related to classic data mining, which is covered in section 2.2.4.3. Repository mining is a broad definition that encompasses various types of investigations performed on software repositories. A unique characteristic of these methods is that they employ software repositories in order to extract data from a project's history rather than study single instances of a project. In this section, we present methods proposed in the literature to mine software repositories, organised by the type and scope of the results achieved. A more thorough examination of the area is presented by Kagdi et al. in their survey [KCM07b].

Repository mining is a widely used technique for exploratory and analytical studies and currently constitutes a very active topic of research. The following list summarises the application of MSR techniques to study various research problems. The categorisation scheme employed was first proposed in [KCM07b]; we built on it and extended it to include categories not present in the original scheme, as their specific target was software evolution. As in the case of Kagdi et al., the categorisation cannot be strict, as papers touch subjects in various categories. In such cases, We assigned the paper to its major category; Figure 2.3 hopefully disambiguates category overlapping.

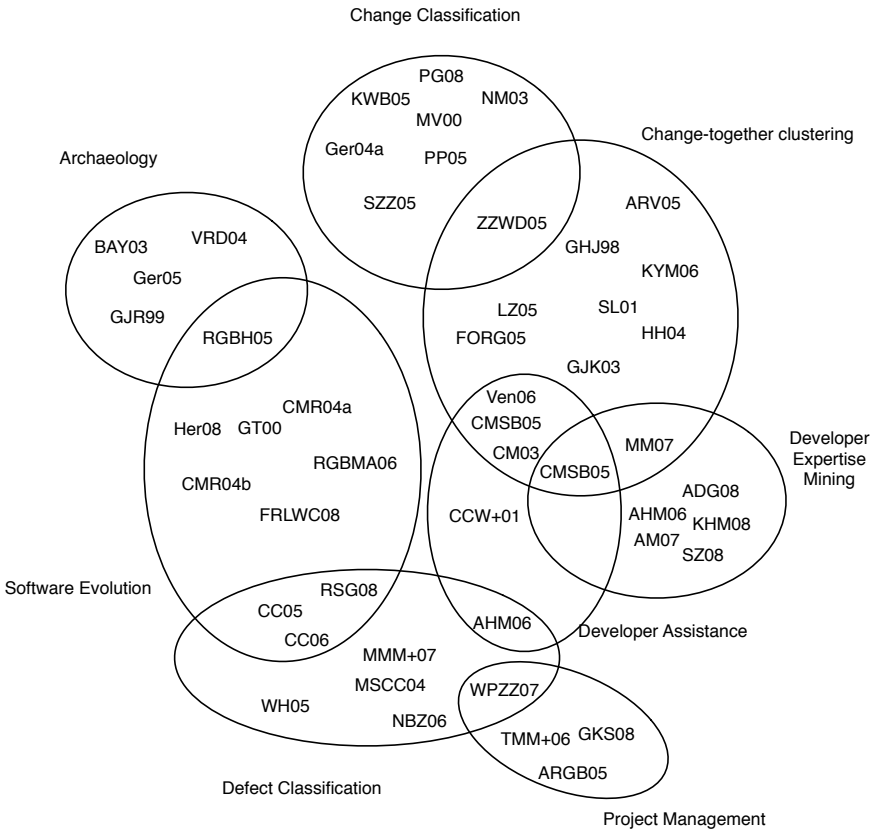

Figure 2.3: Overlap of MSR research categories

Change-together clustering involves the identification of software artefacts that change together. The hypothesis under investigation is that if a cluster of similar objects changed together in the past, they will continue to change together. This helps the identification of re-occurring erroneous behaviour and/or the provision of context help to developers, based on recorded actions performed in prior resource modification sessions. Studies have studied artefacts such as source code files ([SL01, ARV05, ZZWD05, KYM06, MM07]), modules ([GHJ98, GJK03]) or program code ([HH04, FORG05, LZ05]).

Change classification According to the IEEE glossary of software engineering [IEE90], all changes during software maintenance can be classified as either corrective, perfective or adaptive. Several works employed software repositories to extract information in order to automatically classify changes to source code in the above categories [MV00, NM03, PP05] or in clusters according to the change size [PP05], the changed artefact type [Ger04a] or the refactoring function that yielded the change [KWB05]. Corrective maintenance action identification is studied in detail in [SZZ05]. More fine grained change analysis can also be done using compiled

source code [PG08].

Software Evolution Studies of software evolution are concerned with the analysis of the properties that govern the growth (or the lack thereof) of a software system. The foundations of software evolution research were set by Lehman in the form of 8 laws [LRW+97]. Software repositories are used to extract measurements of a project's source code size [CMR04b, GT00, Her08], source code module size [CMR04a, FRLWC08] or even cumulative measurements of various projects [RGBMA06] at various points of their lifetime in order to generate models or predictions of how software will evolve.

Defect Classification Software repositories were used in studies to correlate issue reports with software structure measurements [MSCC04, NBZ06], to guide static analysis [WH05a], to assign bugs reports to developers in an automatic fashion [AHM06], to study the impact of bugs reports on software evolution [CC05, CC06, RSG08] and to group together similar bug reports [MMM+07] in order to discover duplicates.

Social Network analysis As a technique, Social Network Analysis (SNA) is described in section 2.2.4.2. Software repositories have been used to extract data in order to analyse the community structure [BLR04, Ger06], to build social networks and analysing interactions from emails [BGD+07, BGD+06, VTG+06] or from commit messages [BM07] and to study knowledge acquisition and proliferation [LM03, SSA06, WM07].

Developer Assistance Software repositories contain a wealth of information about the process that was used to develop the product. Researchers have considered using this information for annotating patches with SCM system log messages [CCW+01] in order to improve the developer's comprehension. Moreover, software repositories where used to construct a project-wide knowledge base consisting of changed-together artifacts and developer actions [CM03, CMSB05, Ven06] in order to minimize the developer's working context.

Archaeology Software archaeology deals with the study of the properties of software that is obsolete and/or unmaintained [HT02]. The theoretical foundations of software aging are described by Parnas in reference [Par94]. Robles et al. [RGBH05] describe a model for identifying aged software while other authors introduce visualisations for discovering unmaintained areas of code [GJR99, BAY03, VRD04, Ger05].

Developer expertise mining Developer expertise mining approaches try to quantify

Table 2.4: Types of social networks in OSS development

Type	Purpose	Works
Developers Network	Study the participation of developers in projects	[MFT02] [XGCM05], [OOOM05][GM07]
Committer Network	Network of developers that have changed the same source code module	[LRB06] [MRRGBOP08]
Module Network	Network of source code modules that have been modified by the same developer	[LRB06]
Communication Network	Network of people that have communicated by exchanging messages on a communication medium	[HM03] [Mut04] [VTG+06] [BGD+06] [SSA06] [CSX+06]
Community Network	Links projects with common developers (or community members)	[WM07] [OOOM05] [VTG+06] [GM07]

the experience a developer has accumulated by working with certain types of artefacts and to build classifiers that recommend developers for future maintenance tasks. Specifically, approaches include analysing commit activity to identify which developer has performed most changes on sets of related files [MM07, ADG08, KHM08], building networks of developers that worked on the same file [SZ08], and mining information from bug reports about who fixed bugs in a specific file [AHM06, AM07]. A system that integrates all MSR data sources in a comprehensive recommendation framework is described in [CMSB05].

Project Management Software repositories store information about the development process that might be of use for project management purposes. This information has been used to estimate effort based on existing team competencies [WPZZ07, TMK+06] and to develop models of developer turnover to development process[ARGB06, **?**].

2.2.4.2 Social Network Analysis

Social networks are graph representations of social structures where each actor is represented by a graph node and its dependencies with other participants are represented by graph edges. Various types of dependencies exist: social networks have been built to model dependencies such as values, visions, work relationships, trade or idea exchanges. SNA is a set of techniques that analyse the structure of social network graphs in order to gain insight on what drives the creation of relationships or how similar graph groups are clustered [WF94].

In the context of software engineering, SNA has been used primarily to understand the structure and operation of development communities. In a comprehensive study of the field [Sow07], Sowe compiled a list of types of social networks that have been used to model certain aspects of OSS development, which can be seen in Table 2.4.

2.2.4.3 Data Mining

The term data mining (or Knowledge Discovery in Databases – KDD) refers to a collection of techniques for extracting knowledge from large sets of structured data. Data mining tasks typically attempt to predict the future behaviour of the dataset by building models from it or to describe the dataset by discovering relationships among the various variables that comprise it [FPSS+96]. A more detailed classification of the techniques involved in data mining is presented below:

Clustering or unsupervised learning algorithms attempt to group similar data objects together. In unsupervised learning scenarios, there are no predefined classes or bias with respect to the valid relations. Data clustering is also used to automatically identify data outliers in data. In a sense, unsupervised learning can be thought of as finding patterns in the data above and beyond what would be considered unstructured noise.

Clustering is a common technique for analysing results in empirical studies; it is used when researchers need to find relationships between seemingly independent variables. Clustering has been used to identify artefacts in software repositories that change together (as described in Section 2.2.4.1), to find defects by analysing program execution profiles [DLA01] and to group together components with similar metric results [KDTM06].

Classification or supervised learning is a data mining technique used to predict group membership for data instances. The classification problem can be defined as: given a collection of records (training set) and a set of predefined categories, induce a model that can be used to separate new data into those categories, with minimal error. Popular classification techniques include decision trees, neural networks and support vector machines.

Classification techniques are in widespread use in research with empirical data, as they allow researches to automatically build models that organise data according to multiple characteristics, thereby increasing data behaviour analysis and prediction capabilities. Classifiers have been used to determine automated testing scenarios [LFK05], to organise projects according to hosting site properties [BRM04], to classify software defect reports [PMM+03, FLMP04, MINK07], to detect coupling between components [GHJ98], to detect maintenance changes, as described in Section 2.2.4.1, and to predict defects based on previous behaviour [MPS08].

Association rule extraction Association rules reveal underlying correlations between the attributes in a data set [AIS93]. Association rules are usually denoted as

$A \rightarrow B$ where A and B are classes of attributes in the data (or columns in relational formats). A mined association rule has two important properties: support defined as the probability of A and B occurring together ($P(A\&B)$) and confidence that signifies the number of appearances of the association with respect to the number of appearances of the left attribute ($P(B|A)$). Association rule mining aims to extract interesting correlations, frequent patterns, associations or casual structures among sets of items in transaction databases or other data repositories. Association rules are widely used in various areas such as telecommunication networks, market and risk management and inventory control.

As software engineering data are progressively stored in standardized relational schemata, association rule extraction will be a promising analysis technique to infer relationships between data. Pioneering work on the field was carried out by [ZZWD05], who presented perhaps the first use of association rules in his developer assistance tool. Association rules have also been used to identify defects that introduce changes [KZPW06], to correlate defects with staff effort [MMM+07] and to recommend developers for solving issues based on their previous experience [KCM07a] .

2.2.4.4 Clone Detection

A code clone is a code portion in source files that is identical or similar to another. A number of reasons can lead to the introduction of clones: bad coding practices, such as copying-and-pasting code, intentional repetition of a code portion for performance enhancement or unintentional repetition due to standard methods and algorithms used by programmers to solve sets of problems (for example, re-implementation of sorting algorithms) [BYM+98]. Code clones are know to affect program maintainability and for this reason a number of clone detection methods have been proposed in the literature [BvDTvE04]:

Text-based techniques [Joh93, DRD99] perform little or no transformation to the source code before attempting to detect identical or similar lines of code. Typically, white space and comments are ignored.

Token-based techniques [Bak95, KKI02] apply a lexical analysis (tokenisation) to the source code, and subsequently use the tokens as a basis for clone detection.

Abstract Syntax Tree (AST)-based techniques [BYM+98] use parsers to first obtain a syntactical representation of the source code, typically an AST. The clone detection algorithms then search for similar subtrees generated by other source code entities. Moreover, program dependence graph approaches [KH01, Kri01]

Table 2.5: Publication outlets considered for this systematic review

TSE	IEEE Transactions on Softare Engineering
TOSEM	ACM Transactions on Software Engineering and Methodology
ICSE	IEEE–ACM International Conference on Software Engineering
EMSE	Empirical Software Engineering Journal
MSR	Working Conference on Mining Software Repositories
OSS	Open Source Systems Conference

go one step further in obtaining a source code representation of high abstraction. Program dependence graphs contain information of semantical nature, such as control and data flow of the program.

Metrics-based techniques [MLM96] are related to hashing algorithms. For each fragment of a program the values of a number of metrics is calculated, which are subsequently used to find similar fragments.

2.3 Analysis of Related Work

2.3.1 A Classification Framework for Empirical Studies

The goal of the comparative literature review presented in this section is to analyse the current practice in performing empirical studies in the software engineering field from a method and tool point of view. The examination is, therefore, going to be normative rather than descriptive and exploratory rather than confirmatory as our aim is to classify the contemporary literature into a set of pre-defined categories. The specific questions we are trying to answer with this study are the following:

- What methods do researchers use in empirical studies?
- What research topics are of interest in recent empirical studies?
- What are the characteristics (data sources, size) of the empirical studies?

To conduct the review in a systematic way, we used the Systematic Review (SR) framework by Kitchenham [KPP+02, Kit04]. The systematic review process is composed of three steps: (1) Planning the review where the researcher identifies the data sources, inclusion criteria and classification categories, (2) Conducting the review, where the data extraction and synthesis takes place and (3) the reporting phase.

The classification started by defining the list of data sources to be used. A list of venues and publication outlets that usually publish empirical studies was compiled, based on experience and consultation. The list can be seen in Table 2.5. Not all data

sources publish studies of the same quality level; this fact had to be accepted for the sake of completeness.

Each data source was scanned for current publications; as current, we considered all works published from 2003 onwards, since other works [GVR02, SHH+05] have already covered earlier literature and our particular focus was on novel tools and methods, so emphasis had to be given on newer studies. We downloaded all current papers that *appeared* (from their title or abstract) to be, or contain results based on, the application of at least one of the empirical research methods outlined above. The emphasis on the word "appeared" in the previous sentence is placed in order to draw the reader's attention to the fact that we did not use any automated search or pattern-word based method to filter the papers consider for review. The total number of papers reviewed grew to about 200. We estimate the total number of works published in the identified publication outlets since 2003 to about 600, so our analysis includes about 1/3 of those.

After the initial acquisition of papers, we studied each one of them. Several works (more than 30) were immediately excluded from the study as they did not match the quality expected from an empirical study: common quality deterioration factors were the absence of clearly stated research questions or the absence of analysis of the research results. The majority of those papers came from the early MSR and OSS conferences. As one might expect, no papers were rejected due to methodological errors from those downloaded from the top tier journals and conferences in our list.

The classification scheme was constructed to provide direct answers to the questions posed above. Specifically, the question about the research methods was transfered directly to the classification scheme. Following the categorisation of study goals presented by Basili in reference [Bas96], we broke down the research topic question in three components; The object of study and purpose of study fields can only take the same values as in [Bas96]. On the other hand, the focus of the study is a free form text field, which is filled in with a short description of the software engineering property under investigation. Finally, the question about the study properties is answered through the remaining three fields of the classification model. All three fields can hold multiple values as it is common for a study to employ more than one data sources and analysis methods. The classification model fields along with the possible values they may contain are presented in Table 2.6.

2.3.2 Analysis

From the 170 papers that were left in our sample after the initial quality cleanup, we randomly selected 70 on which we applied our analysis framework. We feel that due to the randomness of the sample the quality of the works in our study is representative of the quality of the works an empirical software engineering researcher will face when

Table 2.6: A classification framework for empirical software engineering studies

Category	Description	Values
Research Method	The research method used to perform the study.	Survey, Experiment, Case Study
Object of Study	What is assessed by the study?	Process, Product, Model
Purpose of Study	The reason the study is being performed wrt to the object of study	Characterize, Evaluate, Predict, Control, Improve
Focus	What properties of software engineering does the study analyse?	Depends on the study
Analysis Method	The analysis method(s) used for obtaining the results	Data Mining, Statistical Analysis, MSR, SNA, Clone Detection
Data source	Where do the data presented in the study come from?	Source (code), SCM, Electronic Communication Trail (ECT), BTS
Sample size	Number of analysed projects	Natural Number

attempting to investigate the area.

The classification scheme proved easy to apply to our study members. The values that for each one of the classification fields came from the corresponding section in this chapter. Table 2.7 presents all the possible values for each field. The "focus" field proved particularly challenging to populate; initially, we tried to summarize the paper's direction of study in no more than 4 words. As perhaps expected, at the end of the classification effort, this field did not contain any particularly interesting (for our study goals) information; therefore we decided to group together similar focus points into broader categories. The criteria for the classification of focus points into the identified categories were arbitrary, and we iterated over the classification scheme several times. The graph in Figure 2.4 presents an expanded view of the identified categories. We used the labels that are attached to each graph entry to tag entries in the classification table (Table 2.8). We also produced a graphical overview of the frequency of each classification entry, which can be seen in Figure 2.5.

2.3.3 Results

From this comparative literature review study we can extract several useful results, the most important of which is that the average work in empirical software engineering examines the validity of a method or tool by applying it to data coming from few projects. To us, this result was a real surprise, if not a shock: given the vast amounts of data available (Mockus [Moc09] reports amassed data sizes in the order of terabytes from two hundred thousand projects) and that these data are free to use, why do researchers use such limited datasets? One might argue that even small datasets are enough to validate a hypothesis. This is, of course, a valid proposition. However, the

Table 2.7: Legend of possible values for Table 2.8

Research Method

Acronym	Description
CCS	Confirmatory Case Study
ECS	Exploratory Case Study
EXP	Experiment
FCS	Field Case Study
SUR	Survey

Object of Study

Acronym	Description
PRC	Process
PRD	Product
MDL	Model

Purpose of Study

Acronym	Description
CAR	Characterize
CTR	Control
EVL	Evaluate
IMP	Improve
PRD	Predict

Analysis Method

Acronym	Description
DM	Data Mining
MSR	Mining Software Repositories
SNA	Social Network Analysis
INS	Inspections
MTR	Metrics
QST	Questionnaires
VIS	Visualisation
STA	Statistical Methods

Data Source

Acronym	Description
BTS	Bug Tracking System
ECT	Electronic Communication Trails
SCM	Software Configuration Management
SRC	Source Code
PRE	Pre-processed datasets

Figure 2.4: Decomposition of identified research focus categories to individual research focus items.

value of a method can only be reinforced if the method is successfully validated against several datasets and moreover if the method proves fit (in the Darwinian sense) it will be more easily disseminated. This is also observed by Zannier et al. [ZMM06] and the explanation they provide is that researchers are perhaps under pressure to publish new work (also supported by Saw [Sha03]). In addition to their view, we believe that the dataset format disparity and the sheer data volumes are what deter researchers from evaluating their hypothesis more rigorously.

Another interesting finding is that in our sample, we have not found any study that replicates prior work. Given the recognised quality of the venues we considered, we expected that at least a few studies would be devoted to the task of externally validating important works. Even the highest cited works in our study did not receive any external validation. This either means that we either take other researcher's work

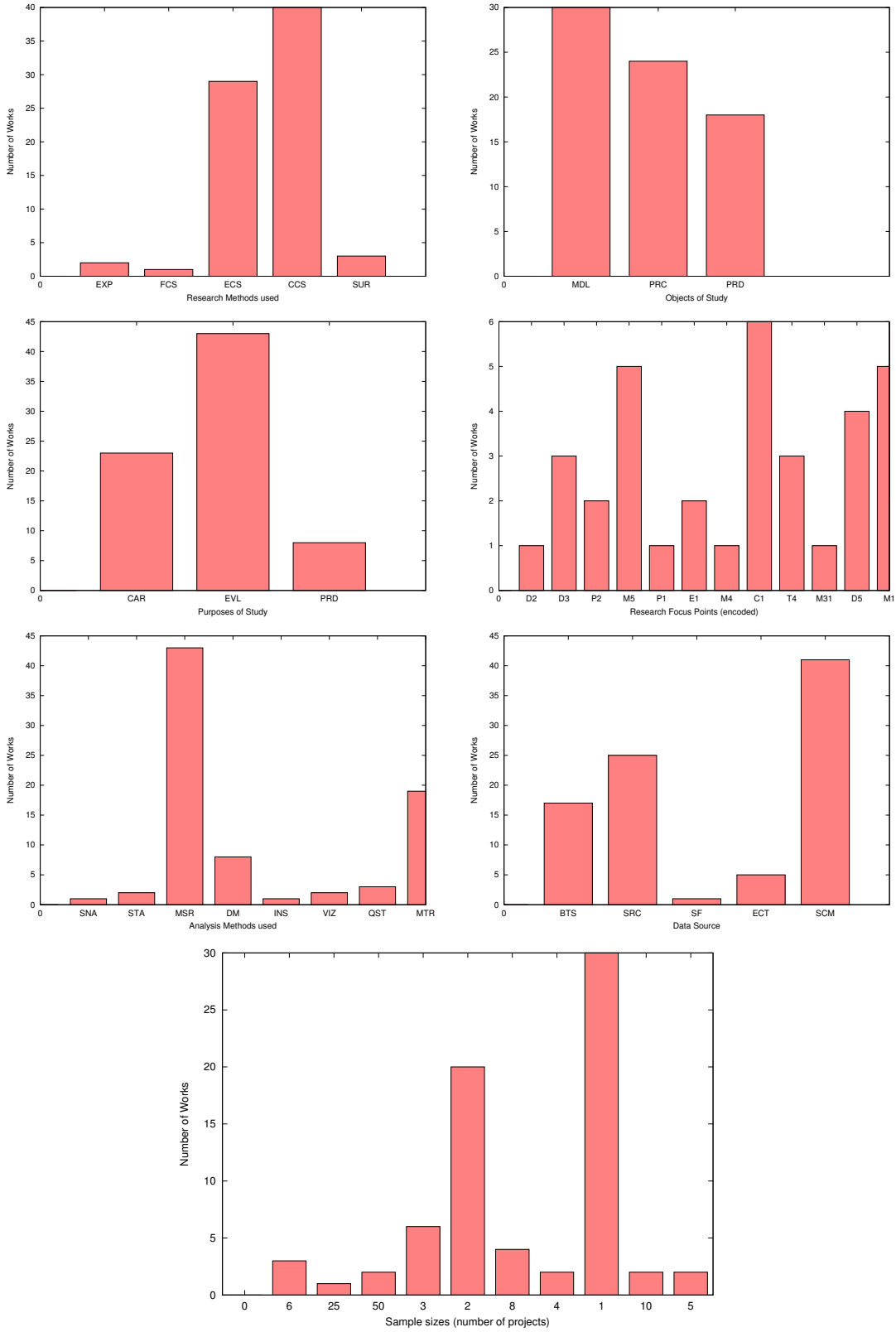

Figure 2.5: Distribution of research methods, objects of study, purposes of study, research focus points (enconding as per Figure 2.4), analysis method, data sources and number of projects in the study sample.

for granted or that we do not build on top of other's work. A possible explanation to this phenomenon was given by Carlo Ghezzi in his keynote address at the ICSE 2009 conference: in a, currently unpublished, study of the full set of proceedings of the ICSE conference, Ghezzi found that only 20% of the tools and data that were reported as vehicles for validation hypothesis could be retrieved and *installed*. This means that researchers do not make it easy to other researchers to replicate their findings, which would work in favour of their published methods.

On the positive side, we see that empirical studies increase over the years and a larger diversity of methods are tested against real world data, which is the preferable approach to validate hypothesis [Bas96, Sha03].

2.4 Summary

Software engineering is an empirical research field, as both new process models and new analysis algorithms need to be validated against existing data. A variety of metrics have been proposed for source code data, even though a small percentage of them is used since the predictive and analytical power of the majority has not been validated in practice. In the recent years, software engineering research benefited from the availability of OSS repositories and a new stream of research that takes advantage of the rich process data residing in those repositories emerged. However, the analysis of the available work shows that the majority of those studies and the methods they produce are not properly empirically validated (limited exposure to real world data, repository-specific techniques) and consequently they should not be generalized.

Table 2.8: Categorisation of empirical studies according to the framework presented in Table 2.6. A legend for field values can be seen in Table 2.7

Work	Research Method	Object	Purpose	Focus	Focus Code	Analysis Method	Data Source	Sample Size
[SJW+02]	CCS	PRD	EVL	Maintainability	M1	MTR	SRC	1
[MVL03]	SUR	MDL	EVL	Defect Indicators	D1	STA	SRC	1
[SSAO04]	ECS	PRD	EVL	Maintainability	M5	MTR	SRC	5
[GJK03]	CCS	MDL	EVL	Module Coupling	M1	MSR	SCM	1
[NM03]	CCS	MDL	EVL, PRD	Evolution	E5	MTR	SRC	1
[BAY03]	CCS	MDL	EVL	Change Proneness	C1	MTR	SRC	1
[MWZ03]	CCS	MDL	EVL, PRD	Effort estimation	P3	MSR, MTR	SCM, SRC	1
[HM03]	SUR, ECS	PRC	CAR	Global Software Development	P2	QST, MSR	SCM	1
[DJ03]	ECS	PRC	EVL	OO Metrics Maintainablity	M1	MTR	SRC	1
[PMM+03]	CCS	PRC	PRD	Failure Report Classification	D1	DM	BTS	2
[GM03]	CCS	PRC	EVL	Metrics gathering tool	T1	MTR	SCM, SRC	1
[FPG03]	CCS	PRC	EVL	Metrics gathering tool	T1	MTR	SRC	1
[CM03]	CCS	MDL	EVL	Project memory	T4	MSR	SCM, ECT, BTS	1
[FLMP04]	CCS	MDL	EVL	Bug classification	D1	DM	BTS	1
[CMR04a]	ECS	PRC	CAR	Stuctural Evolution	E5	MSR	SCM	25
[OW04]	CCS	MDL	PRD	Metrics to predict bugs	D5	MSR, MTR	SCM, BTS	3
[Ger04a]	ECS	PRC	CAR	Modification requests	M5	MSR, VIZ	SCM	6
[PSE04]	ECS	PRD	EVL	Evolution OSS vs closed	E5	MSR, MTR	SCM	6
[CMR04b]	ECS	PRC	CAR	Structural Evolution	E5	MSR	SCM	1
[BvDTvE04]	ECS	MDL	EVL	Cross cutting consearns	M5	STA	SRC	1
[GVG04]	ECS	PRD	CAR	Evolution	E4	MTR, MSR	SRC	1
[RRM04]	ECS	PRC	CAR	Effects of Documentation	M4	MSR	SRC	1
[FSG04]	CCS	PRD	EVL	Metrics tool	T1	MTR	SRC	1
[HH04]	CCS	PRC	EVL	Evolution (Change propagation)	E5	MTR	SCM	5
[MVL04]	EXP	PRD	CAR	Code Smell	M5	QST	SRC	1
[RRL+04]	CCS	PRC	EVL	Low level change tool	T3	MTR	SCM	2
[VRD04]	ECS	PRC	EVL	Evolution visualisation	E2	MSR	SCM	1
[YMNCC04]	CCS	PRC	CAR	Change Prediction	T2	MSR	SCM	2
[BWKG05]	CCS, EXP	PRD	EVL	Software analysis tool	T3	MSR	SCM	8
[Ger05]	ECS	PRC	CAR	Evolution	E1	MSR	SCM	2
[KWB05]	ECS	PRD	CAR	Change Patterns	C1	MSR	SCM	8
[KN05]	CCS	PRD	EVL	Code Cloning	T2	MSR	SCM	2
[LZ05]	CCS	PRD	EVL	Method change patterns	C2	MSR	SCM	2
[RGB05]	ECS	MDL	EVL	Developer identification	DE2	MSR	SCM, BTS, ECT	50
[RGBH05]	ECS	PRD	CAR	Software Aging process	E4	MSR	SCM	8
[Mis05]	ECS	MDL	PRD	Maintainability (design)	M1	MTR	SRC	50
[LYY+05]	CCS	MDL	EVL	Software bug patterns	D1	DM	BTS	4
[LFK05]	CCS	MDL	EVL	Test case failures	P5	DM	SRC	2

Continued on next page

Table 2.8 – continued from previous page

Work	Research Method	Object	Purpose	Focus	Focus Code	Analysis Method	Data Source	Sample Size
[SZZ05]	CCS	PRC	CAR	Fix inducing changes	C1	MSR	SCM, BTS	2
[WH05b]	CCS	PRD	CAR	Extract API usage patterns	C2	MSR	SCM, SRC	1
[BN05]	CCS	MDL	EVL	Clustering artifacts	C4	MSR	SCM	2
[GW05]	CCS	MDL	EVL	Refactoring identification	M31	MSR	SCM	2
[CMSB05]	CCS, FCS	MDL	EVL	Context help	T4	MSR	BTS, SCM, ECT	1
[PP05]	ECS	PRC	CAR	Small change characterisation	C1	MSR	SCM, SRC	1
[WH05a]	CCS	MDL	EVL	Bug identification	D4	MSR	SCM	2
[ZZWD05]	CCS	MDL	EVL	Change together artifacts	C4	MSR	SCM	8
[GFS05]	ECS	PRD	PRD	Metrics predict bugs	D5	MTR	SRC,BTS	2
[CC06]	CCS	MDL	EVL	Response to Issure request	P1	MSR	SCM,BTS	2
[KYM06]	CCS	PRC	EVL	Change together files	C3	MSR	SCM	10
[KPW06]	ECS	PRD	CAR	Design pattern evolution	C1	MSR	SCM	3
[NBZ06]	CCS	MDL	PRD	Metrics predict failures	D5	MSR	SRC,BTS	6
[RGBM06]	ECS	PRD	CAR	Software artifacts evolution	E1	MSR	SCM	1
[Spi06a]	ECS	PRC	CAR	Distributed development	P2	MSR	SCM	1
[SSA06]	ECS	PRC	CAR	Knoweledge distribution	P4	MSR	ECT	1
[AM07]	CCS	MDL	EVL	Developer Expertise (bugs)	DE1	MSR	BTS	1
[BM07]	CCS	MDL	EVL	Match BTS with SRC	D4	MSR	ECT,SCM	2
[MM07]	CCS	MDL	EVL	Bug Squad Recommendation	D3	MSR	SCM,BTS	3
[MMM+07]	ECS	PRC	CAR	Defect Classification	D1	DM	BTS	1
[WPZZ07]	CCS	MDL	EVL	Defect Recov. Time Estimation	D3	DM	BTS	1
[GM07]	ECS	PRC	CAR	Analysis of SF.net Developers	DE	SNA	SF	1
[Moc07]	ECS	PRD	CAR	File reuse	E3	MSR	SCM	10
[ADG08]	CCS	MDL	EVL	Developer Expertise	DE1	MSR, VIZ	SCM	1
[WND08]	CCS	MDL	EVL	Patch classification	C1	MSR	SCM	2
[MPS08]	CCS	PRC	PRD	Defect Prediction	D5	DM	SCM, BTS	3
[SRB+08]	SUR	PRC	CAR	Defect Annotation Usage	D1	QST		2
[GRW08]	ECS	PRD	EVL	Maintainability with ORM	M1	MTR	SRC	2
[FCS+08]	ECS	PRC	EVL	Design Stability	M5	MTR	SRC	2
[WCN08]	ECS	MDL	EVL	Defect Estimation	D3	INS	SRC	2
[Spi08]	ECS	PRD	CAR	Code Quality		MTR	SRC	4
[WZX+08]	CCS	MDL	EVL	Duplicate Defect Report	D2	MSR	BTS	2
[SJM08]	CCS	MDL	EVL	Framework Evolution Detection	E5	DM	SRC	3
[DR08]	CCS	MDL	EVL	Framework Change Recomendations	T4	MSR	SCM	3

Chapter 3

Problem Statement and Proposed Solution

It is a good morning exercise for a research scientist to discard a pet hypothesis every day before breakfast. It keeps him young.

— *Konrad Lorenz*

In the previous chapter, we presented an overview of the field of empirical software engineering. In this chapter, we analyse the shortcomings of current research approaches and provide the foundations of the proposed solution.

3.1 Performing Large Scale Studies with Empirical Data

Research with empirical data is deemed by many as a very important aspect of software engineering [Bas96, WW00, PPV00, SHH+05]. A dedicated research branch has emerged lately (MSR) to take advantage of the abundant information stored by software development process management tools. Moreover, OSS plays an ever increasing role in software engineering research, either as a rich source of process and product data or as a social phenomenon with consequences on team organisation and management in modern projects. But how does the empirical software engineering discipline use all recent advancements in order to evolve?

The scientific method is a Darwinian process in what concerns theory validation: the fittest theory is bound to out-survive the weaker ones. A fit theory is one that invites falsification; in fact, the fittest a theory, the more open to falsification it will be and, at the same time, the more phenomena it will explain [Pop35]. In empirical sciences, scientists do not describe phenomena verbally; they collect measurements and try to put their theories into frames that explain the collected measurements, i.e. they

build models. Models are rejected when they are found incapable of explaining physical behaviours as recorded by measurement data. Exposing a model based theory to public scrutiny is essential for progress in empirical sciences, as the more data a model is tested against, the more likely it is to be found incapable of explaining phenomena.

Our systematic review has shown that the current typical empirical research paper consists of a confirmatory case study that validates the applicability of a model on limited quantities of data originating from SCM systems. Certainly, a model can be validated by exposing it to empirical data. However, there are two (methodological) questions that emerge:

- How can a model be proven fit if it is only tested against data from a single project?
- How can other theories build on a theory that has not yet proven fit?

One might argue that the selection process required to distinguish fit from unfit theories has a time dimension, meaning that if after a long time period that a theory is being used it has not been invalidated, then the theory is probably fit. The argument is partially correct; indeed most theories that stand the test of repeated validations through time (with all changes in observation techniques and observed phenomena time entails) could be fit. However, there is a classic counter-example in software engineering that invalidates the "time-tested theory" conjecture: Halstead's software science theories have been equally invalidated [FL78, Wey88, CALO94, FP98, MSCC04, Her08] and validated in various studies, while in the mean time other theories [CALO94] have been build on top of them. Are Halstead's theories fit or unfit?

Our thesis is that software engineering theories must be invalidated as soon as possible, certainly before they are presented to the public, and invalidation can only be done by rigorously exposing the theory to empirical data. Contrary to common practice of testing models on a limited set of data just to prove that they work, we believe that it is the duty of researchers to try hard to *disprove* their theories before they publish them. If theories are published without being rigorously validated, they can have a corrosive effect on the theories that build on them. Of course, no amount of experimental validation can prove that a theory is correct, but it is still desirable and preferable to have a theory thoroughly tested, especially since empirical data from various distinct sources can be used to test a large set of theories. Moreover, researchers should work towards inviting other researchers to *replicate* their experiments by making the experimentation instruments, namely the tools they developed and the data they used, available to the public.

To return to the question we posed in the introduction of this section, from the systematic review we present in Chapter 2, it appears that, apart from a few studies,

mainstream research presented in major publication outlets exhibits the same characteristics described by Tichy et al. [TLPH95] and Zelkowitz and Wallace [kwW97], namely incomplete experimentation. The situation has certainly improved, as the vast majority of the works include experimental validation of the presented systems and theories and have started taking advantage of data in software repositories. However, very few works rigorously validate their hypotheses with data from more than one data sources.

3.2 The Software Engineering Research Platform

The main problem we are trying to tackle is how to enable software engineering researchers to utilise the vast quantities of freely available data efficiently while allowing fast experiment turnover and experiment replication.

Currently, software engineering researchers have the luxury of access to large volumes of freely available data, thanks to the OSS movement. However, taking advantage of these data is not easy: OSS projects use various combinations of tools for managing SCM repositories, mailing lists and issue databases and consequently raw data are of varying formats. The size of the data sources is another obstacle; a lot of projects have amassed several gigabytes worth of data in their repositories during their lifecycles. Collecting and preprocessing data, calculating metrics, and synthesizing composite results from a large corpus of project artifacts is a tedious and error prone task lacking direct scientific value. Moreover, experience shows that researchers do not share tools and keep re-inventing the wheel. For example, there are several individually developed implementations of the CK metrics suite, that work in various languages and produce varying results, mainly due to liberal interpratation of what the original metric descriptions meant. This segmentation of effort not only affects developers, but also makes experiments non-reproducible. An overview of the situation described above can be obtained by studying Robles's PhD thesis [Rob05].

We believe that this situation can change. What we suggest is an open platform specifically designed to facilitate experimentation with large software engineering datasets derived from diverse data sources, named Software Engineering Research Platform — SERP.[1] Our idea revolves around 4 basic principles (Figure 3.1):

- Rigorous testing of software engineering research artefacts (theories, models, tools) with empirical data should be a prerequisite for research results to be made public.
- Large volumes of empirical data do exist and are available for free.

[1] Throughout this thesis, we use SERP interchangeably as both the name of the platform and as an acronym.

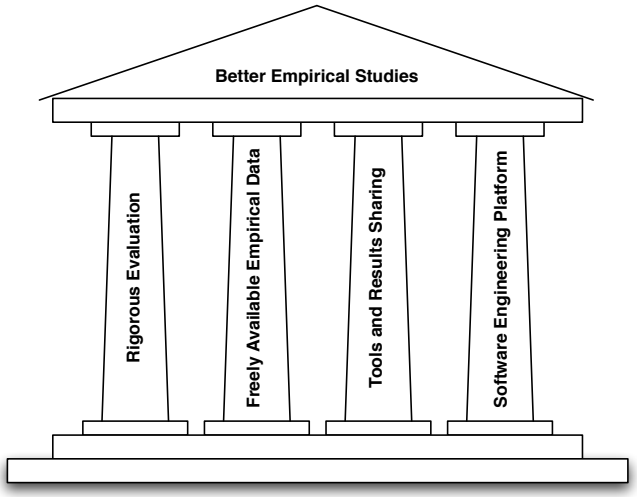

Figure 3.1: The four pillars of better empirical studies

- Sharing of research tools and results accelerates research innovation.
- User ecosystems are formed around software platforms.

Ideally, we would like to build a freely distributable, OSS platform that includes implementations of all known software product and process artefacts analysis and evaluation algorithms and allows the researcher to perform experiments by composing a series of tools programmaticaly. Moreover, it would scale to fully utilise the available computing resources and could provide mechanisms to allow processing results to be shared. If the platform proves successful and is adopted by the research community, then an ecosystem will form around the platform that will share tools, results and methods.

3.3 Hypotheses

The research we propose has the following characteristics:

- It describes a novel system that synthesizes various existing ideas along with new algorithms and purpose-specific representation formats in a coherent framework. Several parts of this work have been described by other researchers, but this is the first time an integrated system of this magnitude is being introduced. Therefore, our working hypotheses only concern the feasibility and the results of the tool we are building and not the methods and techniques that were described by others.
- We introduce new, efficient models for storing and new methods of preprocessing

and analysing data. In these cases, we provide in depth analyses of the algorithms and storage formats developed.

- Our main research outcome is a system that is generic enough to be applied on a variety of research scenarios and which has already proved its value in processing very large datasets. Therefore its success is to be determined not by the outcome of experimental evaluations we describe is this thesis, but rather through performance (space and time) benchmarking and feature comparison with similar tools.

Given the above, this work attempts to validate the following hypotheses, by analysing the design and implementation of a platform targeted to facilitate software engineering research. The hypotheses are:

H1 All major software engineering data sources can be unified under in a common storage and metadata format.

H2 Such a system can provide combined access to the incorporated process and product data sources faster than accessing the data sources with standard operating system tools or with their respective client programs.

H3 Using such a system, case studies can be performed on large datasets more effectively (in terms of ease of setting up and time spend in execution) than by constructing custom tools.

H4 Using such a system, experiments can be replicated equally effectively by researchers other than the ones that initiated the experiment.

3.4 Relation to Other Approaches

Experience has shown that as a research field matures, standardised experimentation environments emerge. By standardised experimentation environments, we mean either shared experimentation infrastructures, shared datasets, reference benchmarks or combinations thereof. This is especially true in software intensive research fields, as software tends to be expensive to produce but cheap to reproduce and share. In this sense, the proposed solution draws ideas from experiences in several other empirical fields: physics researchers and astronomers share laboratory infrastructures (the International Space Station or CERN) and equipment (for example, the Hubble telescope), bio-engineers share large databases of gene information for various species (for example, the Human Genome Project database), while econometrics researchers are usually required to pay in order to be granted access to stock exchange data. In computer science related fields,

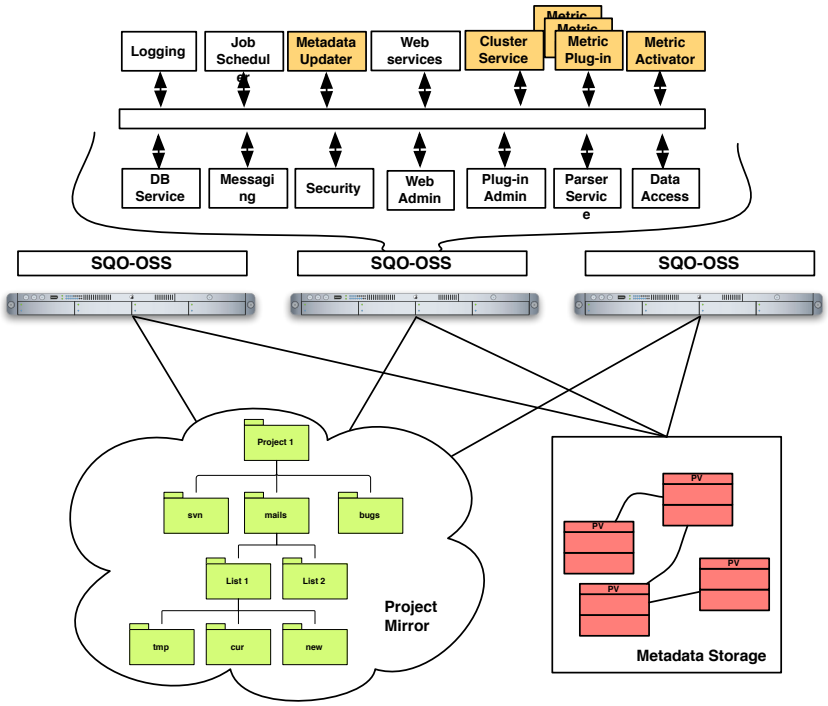

Figure 3.2: The SERP platform architecture with our specific contributions highlighted.

datasets are being shared in the fields of data mining [ACM], bioinformatics while systems such as JikesRVM [AAB00] and ScummVM and benchmarks such as the DaCapo suite [BGH+06] serve virtual machine researchers as testbeds for quickly developing research ideas.

In the software engineering research field, several tools and datasets have been proposed, as we have seen in Sections 2.2.3 and 2.2.2.5. Therefore, one might argue that the proposed approach is neither new nor has any additional value. Such generic statements can be oversimplifying, and as we show, they do not stand. The most important counter argument is that our proposal is first to provide a platform for experimentation instead of a purpose-specific tool. As a platform, we consider both an easily extensible tool and an accompanying dataset. The tool itself is unique as it can automate experiment execution by facilitating the development of analysis tools through powerful abstractions and taking advantage of available hardware. The described dataset is first to combine metadata from three project datasources, by abstracting the underlying data source in a way that allows data from various project management tools to be incorporated, thereby enabling cross-project analysis with no changes in the analysis tools. The platform we propose can (and has) been used for mass scale experimentation, with available hardware being the current bottleneck towards scaling our processed dataset.

SERP is implemented on top of the SQO-OSS tool. Figure 3.2 illustrates the architecture of the SERP platform with a specific focus on how we extended SQO-OSS to implement the proposed functionality. Starting from the data layer, we designed the project mirroring schema (Section 4.2.1) and refactored the metadata database schema (Section 4.2.2) to suit SERP's scaling requirements. Within the SQO-OSS tool, we implemented all metadata updaters (the scientifically important ones are described in Sections 4.4.1 and 4.4.2), we designed the extension interface (Section 4.3) and implemented several platform extensions (for example, the plug-ins described in Chapter 5) along with the extension activation service. Finally, we modified SQO-OSS to run on clusters of machines (Section 4.4.3), and implemented a monitoring service and a load distribution algorithm.

3.5 Limits of Research Scope

In this dissertation we present the design, architecture, and bits of the implementation of a research platform. Even though our work has been based on several existing components, implementing a platform of the scale we propose has been a serious undertaking. For practical reasons, it was decided to leave some aspects of the platform's validation outside the scope of this work and leave the implementation of the corresponding studies as future work. Specifically, we validate the platform's ability to host and process large datasets and to facilitate research through two case studies. However, as it is true with all empirical validations, we only prove that the solution we propose works and is practical; we do not prove whether it is optimal or sufficient for all studies with empirical data. The real limitations of our approach can only be discovered by using the proposed platform in various research scenarios.

A more specific list of limits to this dissertation's research scope is the following:

- We do not provide any means of automatic validation for the metadata and plug-in results. This would entail the construction of a test suite that would automatically compare the data produced by our tools against the data in the project's repository (for metadata) or against data from other well known tools (for metric results).

- In section 4.4.1, we present an algorithm that maps semi-structured SCM data on a fully structured relational schema. As we show, the schema can handle data from both centralised and distributed SCM systems. However, at the time of this writing, there is no implementation of a component to access data in a distributed SCM system programmatically. Implementing such a component would have been very resource intensive for the limited time frame of our research, and therefore

we decided to leave it as future work. As a result, we have only validated the algorithm implementation in centralised repository processing scenarios.

- Even though we have worked to make our platform as generic as possible, we do not provide any validation of the platform's ease of use. We foresee that ease of use will be a major factor for the platform's adoption. We expect this to be reflected in the number of studies that will use our platform.

Chapter 4

Research Platform Design and Implementation

It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.

— *Sir Arthur Conan Doyle*

Platform architectures, as opposed to concrete software implementations, have the following distinctive characteristics:

- They offer extensible data representations to accommodate changes and varying requirements.
- They feature extensibility points for all subsystems.
- They automate tool chain invocations and regulate access to data and to other platform subsystems.

Platform applications is currently a common architectural pattern, that is found in applications ranging from desktop applications (for example, the GNU Emacs and Eclipse) to server applications serving large data volumes, such as the Apache web server and the FaceBook application hosting platform [**?**]. Platform architectures invite sharing of components and code re-use, while they also enable users to modify and extend them to fit their particular needs.

In this chapter, we present a detailed analysis of SERP, a platform specifically targeted to large scale software engineering studies. Specifically, we go through the high level requirements, we present the alternative approaches to meet them and then we analyse the platform's basic components, namely the data it works with and the analysis tools and methods it features.

4.1 Requirements

The requirements for the described platform can be summarised in the following sentence:
A software engineering research platform must facilitate researchers in applying analysis tools on large data volumes originating from diverse data sources while also enabling results sharing.

We analyse each requirement in the following sections.

4.1.1 Integrate Data Sources

An integrated software analysis platform must support the execution of analysis tools on all projects, independent of the product and process data sources in use by the project.

Requirements

- Integrate various data source formats.
- Support multiple process data sources per analysed project.
- Be extensible to new data formats.
- Integrate common data across data sources.

Challenges

- There is a large number of different systems used to manage the development of software. An overview can be seen in Table 4.1. Semantic and operational differences exist even between systems managing the same data source.
- Semantic integration of data across data sources usually depends on project-specific information [MFH02, CC05, CMSB05, CC06, RSG08].
- An integrated software platform must reconcile semantic differences between different systems managing the same data sources, in order to enable cross-project examination and comparisons. For example, while both distributed and centralised SCM systems support the same basic operations there is no notion of project history in distributed SCM systems; in those cases, the platform must devise ways to approximate project history by aggregating commits in various branches on a single timeline.

Table 4.1: Non-exhaustive list of software process support systems

Data Source	Systems
SCM	CVS, SVN, SourceSafe, Perforce, GIT, Mercurial, Bazaar, Arch, Monotone
Issue Tracking	Bugzilla, Mantis, Jira, GNATS
Mailing Lists	Mailman, Nabble, SourceForge (SF).net mail
Documentation	Javadoc, Doxygen, Docbook
Collaboration	MediaWiki, MoinMoin, Xwiki
Instant Communication	`ircd`, SupyBot

- Each system uses its own data storage and data exchange formats. A research platform must be able to abstract the differences while also enabling direct access to the underlying data for analysis tools that request so.

Alternatives

1. Directly use the data from each data source, by means of purpose-specific programs.
2. Build ad hoc, text based intermediate result and raw data formats.
3. Pre-process the data, store intermediate results in a custom database, extract required information for the task at hand.
4. Design a relational format that abstracts the least subset of features for each data source, pre-process data, construct links to the original data and perform the analysis by selecting the appropriate data representation for the occasion.
5. Store all data in an all-inclusive relational schema, and use this schema for processing.

Approach We chose to meet the requirements by designing an intermediate data abstracting, relational format. This choice is driven by two factors: (1) Analysis tools should not know about the underlying repository formats, and (2) it should be easier for platform users to learn one standardized intermediate data format than to delve into the intricacies of each project's data sources.

4.1.2 Manage Computing Resources Efficiently

A software engineering research platform must exploit the available computing resources on a single computer in full, with minimal overhead to the user.

Requirements

- Efficient use of available computing resources.
- Scaling to occupy all available resources.
- Automatic resource management, hidden from the platform user.

Challenges

- Making efficient use of multiple processors is a difficult proposition, as it usually entails converting serial algorithms to parallel. The easiest way of achieving parallelism is to run multiple instances of a serial program on independent data, in which case the problem to be solved is deemed as trivially parallelisable. Fortunately, most experiments in software engineering research are indeed trivially parallelisable, as they usually apply a tool on independent states of a project's data and aggregate selected measurements. Therefore, a platform should optimise for the general case, where the unit of work is an analysis tool applied on a project resource state, and provide mechanisms to parallelize this scenario.
- There exist analysis methods whose results depend on the results of the application of the method on a previous resource state, and therefore the analysis must be executed serially. A typical example is the calculation of the number of developers that are active in a project within a specified time frame; as any project state is the result of the actions of a single developer (for example the one that committed a changeset to a project's SCM system), we must be able to retrieve the developers that were active in the timeframe up to the specified point. This computation cannot be performed in parallel without guarantees on the execution order, or else the results will be incorrect. In that case, the platform must support the expression of dependencies between elements of the computation, and the ordering of elements according to the dependencies.
- Incorporating external tools can affect a platform's ability to manage computing resources. Currently, several analysis tools have the ability to split the load in multiple processors. The platform cannot control the resource management mechanisms of tools that are not written to utilise the platform's internal facilities.

Alternatives

1. Don't manage resources at all; expect researchers to write tools that parallelize the workload as appropriate and trust resource management to be carried out effectively by the operating system.

Metric	Evolution	GVFS	KDE
Years of RCS history	10y11m	9y4m	11y8m
Number of revisions	36835	5522	893049
Number of live files	52931	16712	$> 5 * 10^5$
Repository size	1.4GB	103MB	48GB
Community size	389	653	$> 3 * 10^3$
Number of emails (total)	1303	20	36696
Number of emails (Nov08 only)	240	5	6252
Number of Bugs (collected)	2070	292	15902

Table 4.2: Key size metrics for selected projects as of November 2008.

2. Use process abstractions provided by the operating system in order to schedule units of work on available processors. Implement data ordering policies and data slicing mechanisms and automatically run analysis tools using the appropriate ordering policy and data size.

3. Provide a work unit abstraction mechanism that analysis tools use in order to assign slices of the data to analysis algorithms. Assign each work unit a data slice and schedule the work units on available processors. Allow analysis tools to specify priorities on work units.

Approach We chose the work unit abstraction pattern to abstract analysis tool runs; this pattern effectively separates resource management from process initiation tasks, delegating each task to the part of the system that has the most information and can make the best decisions. The platform core knows the number of available processors and has direct access to the data to be processed while each analysis tool knows the correct ordering of the input data in order to run effectively.

4.1.3 Working with Large Data Volumes

Given the appropriate hardware, a software engineering research platform must support the application of analysis algorithms on arbitrarily large volumes of data.

Requirements

- Work on clusters of machines.
- Scale gracefully to large data sizes and distribute computations.

Challenges

- The majority of free data for empirical studies are available online in the form of OSS projects. Several OSS projects have a history spanning multiple decades

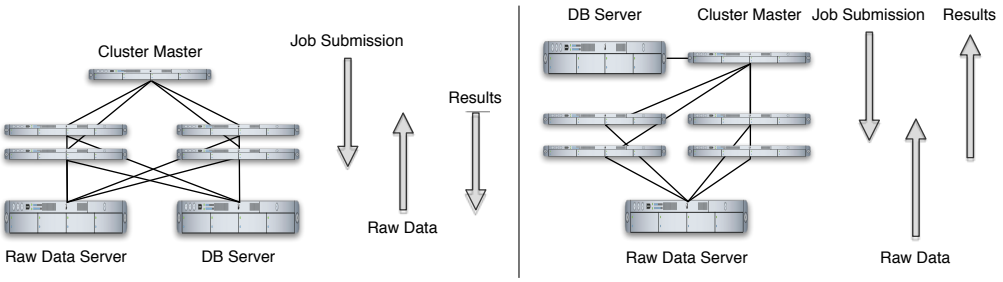

Figure 4.1: Program (left) vs data (right) clustering.

and extremely large software repositories. Table 4.2 presents three such projects. The sizes of the datasets involved are comparable to other empirical datasets in other sciences which already use high performance computation techniques. A software data analysis platform must therefore employ equivalent workload processing schemes.

- Computational latency plays a significant role in performance scaling. A motivating example is the following: An average size for a small to medium project is in the order of 5000 revisions. Each revision of the project can have thousands of live files, a large number of which are source code files. A rough calculation for an average 5KB file size and for an average 500 live files per revision indicates that a system would read 12GB of data in order to load the processed file contents into memory. This shows that the simplest of operations are prohibitively expensive to do online as it introduces large latencies and could potentially hurt the performance of the project hosting servers.

Alternatives

1. Construct purpose made programs that query on-line project resources for data, run analysis algorithms on single computer, store the results in a database.

2. Program-based clustering: use a two tier architecture to split responsibilities for computation and storage, split computational nodes on a cluster and have all cluster nodes update the same database. A master node submits work items to computational nodes, while raw data are shared via a network-oriented filesystem.

3. Data-based clustering: use cluster load distribution techniques, such as Map-Reduce [DG04], to split the data in manageable work items per computational node. All nodes can play the role of the load distributor, but only the load distributor can update the result database with the results of the distributed computation.

Approach We used the program-based clustering paradigm (see Figure 4.1). We believe that program-based clustering is better suited to the computational needs of a platform for analysing software engineering data for the following reasons:

1. It delegates input format abstraction and processing to each cluster node, thereby reducing the load on the cluster master.
2. It delegates the task of result aggregation and calculation to each compute node, which can accommodate better the way the average software analysis algorithm works.
3. It allows the platform to work on a single host for faster development turnover.
4. It can effectively work with analysis algorithms that require some form of serialisation of the workload to process. In this case, parallelism is achieved from running multiple projects on multiple parallel nodes.

4.1.4 Result Sharing and Experiment Replication

A software engineering research platform must enable researchers to share raw data, results and experimental setups and automate the process of exchange.

Requirements

- Facilitate research tool exchange
- Exchange tool results and analysis metadata with minimal overhead.

Challenges

- Before being shared, the results must be stored in a local datastore.
- Programmatic access to the datastore must be implemented. It must enable researchers to get data in an aggregated fashion with simple tools, for example the Unix command line toolchain.

Alternatives

Tool Exchange

1. Share the tools in source code form via a shared SCM repository.
2. Share the tools in binary form, by integrating in the platform mechanisms that automatically publish tools in a shared location and can discover and integrate in the analysis process newly published tools.

Results Exchange

1. Construct pre-computed sets of metadata and make them available over the web.
2. Develop programmatic interfaces to the platform's internal data stores.
3. Develop mechanisms to connect various platform instances in a peer-to-peer or centralised topology and develop protocols to synchronize data among them.

Approach In order to share the experimental results, researchers must exchange the data (raw and metadata) and the tools used to produce them. For tool sharing, we opted to follow a hybrid approach: we build a shared source code repository where interested researchers can share their tools while also enabling our platform to install custom tools from user provided locations. To share the data, we develop programmatic interfaces to the platform's database. Using the described combination of tool and data sharing, experiments can be replicated at the cost of the appropriate hardware platform.

4.2 Data

In the following sections, we analyse the formats that are used as input to the SERP and present our proposal for organising a data mirroring scheme for freely available OSS data. We also elaborate on the intermediate metadata schema and examine its implications on analysis tools.

4.2.1 Raw Data and Mirroring

SERP, can accommodate three types of raw data, namely SCM data, mailing list data and BTS data. The analysis tools that work within SERP should be data format agnostic, however the design of the metadata database schema is strongly influenced by the data formats that SERP handles.

4.2.1.1 SCM Data

SERP utilises data originating from two distinctively different types of SCM data sources: centralised systems, such as CVS and SVN, and decentralised systems, such as GIT and Mercurial. While both types of repositories perform essentially the same function, the philosophy that undermines their operation is vastly different. We will focus our explanation on one representative system of each SCM type, namely SVN and GIT. Earlier, works [Ger04b, ZW04, RKGB04] have examined the use of CVS for SCM analysis, but most OSS projects have migrated their repositories to SVN since. Currently, there is a tendency among the largest of projects to migrate their repositories to decentralised repositories.

SVN treats the SCM repository as a filesystem tree. SVN strives to maintain consistency in the repository; each commit is guaranteed to be atomic (two commits from independent developers cannot affect the same version of a file) while the repository is globally versioned, with each commit increasing the version of the managed file tree. Uniquely, SVN supports copying of versioned objects between locations and versions of the repository while maintaining the object history. This feature enables creating branches and tags by simply copying the main development file tree to other repository locations, with minimal overhead. Recent versions of SVN also support the maintenance of object history when the changes between two tree paths are merged. As a result of the above, SVN is very good at maintaining the history of versioned objects; this makes it an ideal target for software engineering studies, albeit a very sophisticated one for that matter.

On the other hand, GIT is essentially a sophisticated patch management system. The minimal change set in GIT is a patch that has been applied to a branch, and moreover patches can be moved freely among branches. GIT differs from SVN on how it handles branches: branches are created locally and can maintain the full history of patches after the branch is merged with another branch, while it can also use multiple branches as sources for a merge operation. GIT's distributed nature means that development information in non-published branches is lost. More importantly, GIT can also be ordered to cascade several individual changes into a single change, a feature actively used in projects. This means that a commit to a branch might not entirely reflect the actual history of code changes; in practice though, developers use git to commit very often and consequently aggregated patches are usually of the same size as an SVN patch. Most projects using GIT have a central repository where all branches are aggregated.

In a nutshell, SVN is a versioned filesystem while GIT is an advanced patch submission and management system. In typical analysis scenarios, they offer the same information. GIT improves over SVN in what concerns branching and authorship tracking. On the other hand, SVN offers simpler abstractions and stronger guarantees of project history maintenance, even though navigating history backwards may result in loss of information at merge points. With GIT, similar results can be obtained by retrieving contents of all branches and ordering them by their timestamp. Tools exist to support the conversion between both formats, even though the conversion from GIT to SVN will involve loss of merge and authorship information. An examination of the capabilities of GIT for MSR analyses is provided by [BRB+09].

Mirroring both SCM repository types is simple. In the case of SVN, the `svnsync` tool can be used. The tool will create a read-only mirror of a remote project repository; after the synchronisation is completed the mirror repository can be accessed with the standard SCM toolset. Repository synchronisation of GIT repositories is even simpler;

GIT will automatically download the full history of a branch when a branch is pulled from the main project repository. However, a researcher interested in the full project history should also consider alternative branches, as described earlier.

4.2.1.2 Mailing List Data

Mailing lists are used by projects for information exchange. The importance of mailing list in software analysis is described in Sections 2.2.2.3 and 2.2.4.2. Mailing lists work by aliasing a group of developer emails with a single email address. Posts to the mailing list address are distributed to all people in the group. Advanced mailing list management software is usually employed to automate management tasks, such as subscription and message archiving. Most open source projects offer their mailing list archives over the web, through custom web interfaces, while online services exist that archive email exchanges for subscribed lists.

The minimal change entity in a mailing list an email. Emails are ASCII-text encoded files that follow the RFC-822 [Cro82] and RFC-2045 [FB96] formats. RFC-822 defines emails as a key-value pair formated header and a freely formated payload and specifies that email payloads should be encoded using the ASCII plain text encoding. This specification restricted the use of email in non-English languages, so it was superseded by the RFC-2035 standard, which converted the email payload section in a container format. The type of the objects which exist in each MIME-encoded container section is defined in a custom header. An extract of an RFC-822 encoded email is presented in Listing 1.

Mail processing tools are free to append custom headers to emails, which is common practice for tools managing mailing lists. Therefore not all email headers are useful for analysis purposes. The most important email headers are the following:

- `From`: The email address, usually accompanied by a real name, of the person that has sent the email. This field is the only source of raw data in that directly links a name with an email address. It is also useful to identify developers (uniquely or heuristically), whose email has changed during the course of the project.
- `Date`: The date the email was sent. Mainly used for statistical purposes, in order to assess the rate of information exchange taking place within specific days of the week or hours in a day. Its value refers to the time on the sender's workstation when the email was sent and can therefore be inaccurate very often.
- `Message-ID`: A string uniquely identifying each email. Can be used to build inverted indices in order to speed up access to a large volume of emails.
- `In-Reply-To` and `References`: Both equal to the `Message-ID` of the email that this email is a reply to. In some cases (e.g. Usenet groups) the `References`

```
(Standard Headers)

Return-Path: <kenny@stanford.edu>
Delivered-To: gnome-vfs-list@gnome.org
Received: from webmail.Stanford.EDU (webmail.Stanford.EDU 171.64.14.230)
    by mail.gnome.org (Postfix) with ESMTP id 133A51811B
    for <gnome-vfs-list@gnome.org>; Fri, 19 Jul 2002 15:28:11 -0400 (EDT)

To: Eric Cartman<ec@southpark.com>
Subject: Re: Patch: fix openssl include
Date: Fri, 19 Jul 2002 12:28:01 -0700
From: Kenny McCormick <kenny@stanford.edu>
Cc: gnome-vfs-list@gnome.org
Message-ID: <1027106881.3d3868420451a@webmail.stanford.edu>
References: <1027067952.32690.173.camel@trinidad.mandrakesoft.com>
In-Reply-To: <1027067952.32690.173.camel@trinidad.mandrakesoft.com>
MIME-Version: 1.0
Content-Type: text/plain; charset=ISO-8859-1
Content-Transfer-Encoding: 8bit
Sender: gnome-vfs-list-admin@gnome.org

(Custom Headers)
X-Mailer: Stanford Webmail v1.1.6 17-June-2002
Errors-To: gnome-vfs-list-admin@gnome.org
X-BeenThere: gnome-vfs-list@gnome.org
X-Loop: gnome-vfs-list@gnome.org
X-Mailman-Version: 2.0.8
Precedence: bulk
List-Help: <mailto:gnome-vfs-list-request@gnome.org?subject=help>
List-Post: <mailto:gnome-vfs-list@gnome.org>
List-Subscribe: <http://mail.gnome.org/mailman/listinfo/gnome-vfs-list>,
    <mailto:gnome-vfs-list-request@gnome.org?subject=subscribe>
List-Id: <gnome-vfs-list.gnome.org>
List-Unsubscribe: <http://mail.gnome.org/mailman/listinfo/gnome-vfs-list>,
    <mailto:gnome-vfs-list-request@gnome.org?subject=unsubscribe>
List-Archive: <http://mail.gnome.org/archives/gnome-vfs-list/>

(Actual content omitted for brevity)
```

Listing 1: Example email headers

field refers to the top-level message in the thread that the examined message participates to. Both fields are used to reconstruct parent-child relationships among messages in a mailing list.

- `Content-*` and `MIME-*`: Specify the content type and encoding of the message payload. For analysis purposes, plain text is the preferable format, which the exception of MIME messages with plain text content disposition sections, where plain text can be extracted easily. On the other hand, HyperText Markup Language (HTML)-formated emails are particularly difficult to process. Email attachments are also useful for analysis as attachments usually contain patches or debugging information. Techniques have been developed to identify those in plain text or email sources.

Despite the fact that mailing list data are relatively simple to store and process, they are very difficult to obtain. The majority of OSS projects maintain custom web

```
<bug>
    <!--Generic fields-->
    <bug_id>317632</bug_id>
    <creation_ts>2005-09-30 18:52 UTC</creation_ts>
    <short_desc> A new bug </short_desc>
    <delta_ts>2008-09-06 19:07:54 UTC</delta_ts>
    <product>gnome-vfs</product>
    <component>Module: http</component>
    <version>2.12.x</version>
    <rep_platform>Other</rep_platform>
    <op_sys>All</op_sys>
    <bug_status>RESOLVED</bug_status>
    <resolution>WONTFIX</resolution>
    <priority>Normal</priority>
    <bug_severity>normal</bug_severity>
    <reporter>elanthis@awesomeplay.com</reporter>
    <assigned_to>gnome-vfs-maint@gnome.bugs</assigned_to>
    <cc>a9016009@gmx.de</cc>
    <long_desc>
        <who>elanthis@awesomeplay.com</who>
        <bug_when>2005-09-30 18:52 UTC</bug_when>
        <thetext>Please describe the problem:
    </long_desc>

    <!--Project-specific fields -->
    <target_milestone>---</target_milestone>
    <reporter_accessible>1</reporter_accessible>
    <gnome_version>2.11/2.12</gnome_version>
    <gnome_target>Unspecified</gnome_target>
    <initialowner_id>14140</initialowner_id>
    <qa_contact>gnome-vfs-maint@gnome.bugs</qa_contact>
    <cclist_accessible>1</cclist_accessible>
    <classification_id>3</classification_id>
    <classification>Platform</classification>
</bug>
```

Listing 2: XML-encoded bug report.

interfaces to their mailing list archives, while no mailing list archive service offers programmatic interfaces to the stored data. This means that data have to be retrieved by means of custom tools per project. After the archives have been retrieved, synchronisation of local archives with project mailing list can be simply configured by subscribing to the corresponding mailing lists and redirecting incoming emails to the local mirror for the specific mailing list.

4.2.1.3 Bug Tracking Systems

BTS are used to manage incoming issues and enhancement requests. Currently, there are two widely used systems in OSS, namely Bugzilla and the Tracker system offered by the `SourceForge.net` project hosting site. Both offer roughly the same functionality to end users, however Bugzilla is more customizable as its bug reporting forms can be extended with project-specific fields. Consequently, most large, self-hosted projects prefer Bugzilla over alternatives. Moreover, Bugzilla offers a programmatic interface, which can be used instead of the standard HTML-based interface to retrieve

bug descriptions in XML format. For those reasons, Bugzilla is the de facto source of bug-related data in related studies.

An XML version of a Bugzilla bug report can be seen in Listing 2. The most important fields are the following:

BugId A unique identification number for each bug in the project's BTS. Several projects maintain the convention to include this number in SCM commit messages that affect code the bug report refers to, which in turn is important for identifying the effort spent per bug report or the developer expertise in certain system components.

Severity, Priority and **Status** Classification fields for bugs. Used in analyses methods to group similar bugs together and to correlate with other code metrics.

Reporter, or **Who** in bug comments. The person that reported or commented on the bug. Used to link a bug report with the rest of the developer's activity in the project.

Product, Version and **Component**. Identifies the part of the system that the bug report affects. Used in conjunction with pattern matching techniques to relate a bug report with specific code changes.

Timestamp fields are, among others, used to assess how fast the development team responds to incoming reports to fix or to triage them.

Due to the offered programmatic interface, collecting Bugzilla bugs for a project is straightforward, and can be automated with minimal tooling. Moreover, Bugzilla offers a full suite of database querying facilities, which makes it easy both to synchronise bugs in regular intervals, by querying for the bugs that have changed since the last synchronization and to download the bugs that affect only a specific project part. For SERP, we chose to store each bug in its own file on the local project mirror and have created scripts to automate the mirroring of bugs.

4.2.1.4 The Mirroring Schema

The SERP itself is not concerned with mirroring data from projects; it expects data to be mirrored externally. This choice was made at the beginning of the project to compensate for the large number of different data sources that the system should work with. Several of those already provide the means for synchronizing data across sites. For example, SVN offers the `svnsync` tool for mirroring repositories while distributed SCM systems already copy the full repository history on checkout. Mailing lists can be mirrored by configuring the mail delivery subsystem on the mirroring host to store

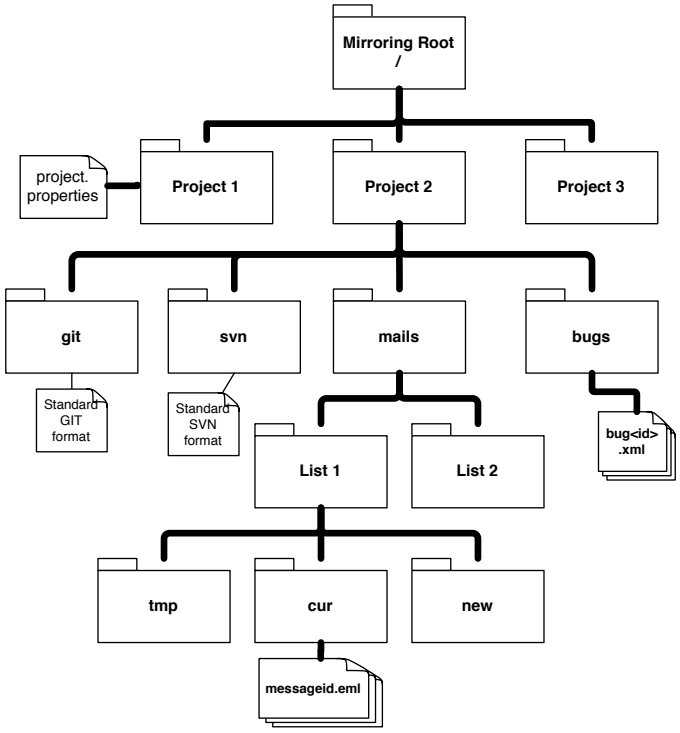

Figure 4.2: The project mirror schema

incoming messages to a specified directory and then subscribing to each mailing list. Mailing lists archive mirroring requires custom scripts per project site, although various mailing list archiving web sites (e.g. MARC) offer a unified view over thousands of mailing lists for common projects. Finally, bug tracking tools usually offer programmatic interfaces to retrieve individual bug histories through programmatic interfaces.

In Figure 4.2, we present an outline of the data mirroring schema we propose. All projects directories are stored under a single directory, the mirroring root. Each project directory contains three subdirectories, one for each mirrored data resource. Since a project can either be hosted in a GIT or an SVN repository only one of the `git` and `svn` directories can be present. All bug reports for a project are stored in a single directory; to speed up access, an alternative format would entail storing bug reports in subdirectories of the top level directory, with a fixed amount of files per sub directory. Finally, email messages are stored in a directory per mailing list; each mailing list directory is formatted according to the Maildir standard. Incoming emails are stored in the `new` sub-directory while processed emails are stored under `cur`.

To minimize configuration when processing the project and to automate the process of importing projects, each project is assigned a configuration file when it is added to the mirroring infrastructure. An example properties file is presented in Table 4.2.1.4.

Table 4.3: The `project.properties` file format

Field	Description
`eu.sqooss.project.bts.source`	The Universal Resource Locator (URL) of the project's Bugzilla
`eu.sqooss.project.bts.url`	The URL of the local bug mirror
`eu.sqooss.project.ml.url`	The URL of the local mail mirror
`eu.sqooss.project.scm.path.excl`	Paths to exclude when processing the project's SCM DATA.
`eu.sqooss.project.scm.path.incl`	Paths to include when processing the project's SCM DATA.
`eu.sqooss.project.scm.source`	The URL of the project's remote
`eu.sqooss.project.scm.type`	The type of SCM updater to use.
`eu.sqooss.project.scm.svn.tag`	Configure the SVN updater to mark subdirectories of this directory as SVN tags.
`eu.sqooss.project.scm.svn.trunk`	Configure the SVN updater to mark the provided directory as the main development tree
`eu.sqooss.project.scm.svn.branch`	Configure the SVN updater to mark subdirectories of this directory as development tags.
`eu.sqooss.project.scm.url`	A URL pointing to the project's SCM repository
`eu.sqooss.project.website`	The project's website (for reference)

The properties file is used by the updater service (see Section 2.2.3.1) to drive the metadata update process.

4.2.2 Structured Metadata

The SERP system uses a database to store metadata about the processed projects. The role of the metadata is not to create replicas of the raw data in the database, but rather to provide a set of entities against which analysis tools work with, while also enabling efficient storage and fast retrieval of the project state at any point of the project's lifetime. Moreover, the storage schema is designed to include the minimum amount of data required in order to abstract the differences between different raw data management systems. The database schema is shown in Figure 4.3. The schema is composed of four sub-schemata, one for each processed data source and one that deals with metric configuration and measurements. The basic entities stored in the database schema are described below:

StoredProject The top-level entity of the SERP platform schema. It represents a project that can have multiple project versions, multiple mailing lists and multiple bug reports.

Developer A developer is a person that has contributed to the project. In the context

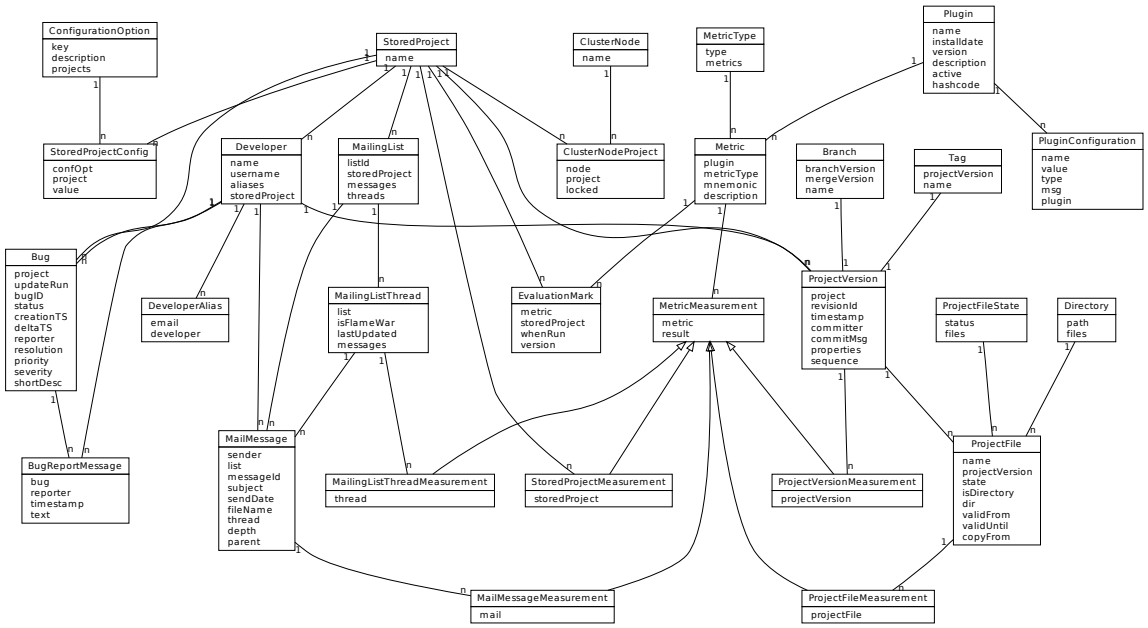

Figure 4.3: The data storage schema

of a single project, developers can be uniquely identified by the email they use to post to mailing lists and submit bug reports or by the user name they use in the project repository. However, in the lifetime of the project, a single developer may have multiple identities, for example because he switched email addresses. In the cases where an email address can be traced to an existing developer, an entry is added to the `DeveloperAlias` table. Later in this chapter (Section 4.4.2), we present an algorithm to resolve developer identities across data sources.

ProjectVersion Represents a state in the project's source code history, as recorded by the SCM system. A new project version is initiated by a `Developer` and contains new instances of one or more `ProjectFiles`. Along with a new `ProjectVersion` a developer can assign a `Tag` to the code or create a new `Branch`; in both cases, an entry is recorded in the corresponding table.

ProjectFile A state in a file's development history. An entry in the ProjectFile table represents either a regular file or a directory. For directories, SERP records their path relative to the project top level directory in the `Directory` table. A file can be generated by the user or copied by another location of the project's file tree; in that case the `copyFrom` field contains a link to the source file. The lifetime of the specific instance of a file is recorded in the `validFrom` and `validUntil`

fields. Using those fields, the system can retrieve a full list of files that are live in a `ProjectVersion`, which is equivalent to checking out the project directly from the SCM system.

MailMessage Represents and individual message sent by a `Developer`, to a `MailingList` and belonging to a `MailingListThread`. Several email header fields, but not the content, are kept for each email to simplify processing and to speed up searching.

MailingList A mailing list represents a collection of `MailMessage`s that are sent to (or carbon-copied to) a common email address.

MailingListThread A thread is another way to organise `MailMessage`s. A thread is a collection of emails that have the same `In-Reply-To` or `References` headers or their `In-Reply-To` or `References` headers point to emails that belong to a thread. By convention, a thread can only belong to a single `MailingList`.

Bug Represents an entry in the project's BTS. Important information such as severity, priority and status that appear in several bug tracking systems are recorded in the schema. A `Bug` can have several `BugReportMessage`s attached to it.

Plugin Represents and holds information about a metric plug-in. A `Plugin` can define several `Metric`s, which are uniquely identified by a *mnemonic*.

MetricMeasurement is the entity that encapsulates all metric results. In SERP, all metric results are bound to one metadata entity, based on the notion that a measurement is the application of a metric on a project resource state. Consequently, for each project entity, there exists a results entity that links it with a metric that represents a measurement.

The role of the metadata schema is pivotal in disengaging analysis tool implementation from raw data formats. Using the metadata schema, analysis tools can process SCM system version logs, project files, mailing list threads and other entities without dealing with the intricacies of each particular system. This enables analysis tools to be written faster and to be compact while also broadening their applicability. The intermediate data approach may not be novel (the RHDB [FPG03], Kenyon [BWKG05] and Hipikat [CMSB05] systems used it at various granularity levels), however this is the first time that an intermediate schema can integrate data from various raw data formats, in a minimal, yet extensible, manner. Later in this chapter (Section 4.4.1) we show how two relatively different SCM systems, namely SVN and GIT, can be used to populate the intermediate schema.

```
public interface AlitheiaPlugin {
  String getVersion();
  String getAuthor();
  Date getDateInstalled();
  String getName();
  String getDescription();
  List<Metric> getSupportedMetrics();
  Result getResultIfAlreadyCalculated(DAObject o,
      List<Metric> l);
  Result getResult(DAObject o, List<Metric> l);
  void run(DAObject o);
  boolean install();
  boolean update();
  boolean remove();
  boolean cleanup(DAObject sp);
  String getUniqueKey();
  List<Class<? extends DAObject>>
               getActivationTypes();
  Class<? extends DAObject>
               getMetricActivationType(Metric m);
  Set<PluginConfiguration>
               getConfigurationSchema();
  List<String> getDependencies();
}
```

Listing 3: The SERP plugin interface. Boxed are the methods that each plug-in must implement.

4.3 Tools

Our work on the SERP, is based on the SQO-OSS tool, described in section 2.2.3.1. We selected the SQO-OSS tool as the basis for our work as it already implemented much of the required functionality and we were familiar with its internals. Our platform and SQO-OSS are closely bound together; in order to benefit from our platform a researcher must first become familiar with SQO-OSS and its internals. SQO-OSS is described in detail in references [GKS+07, **?**, GS09]. Here, we describe how our platform can be extended through plug-ins to SQO-OSS.

The SERP platform can be extended by plug-ins that perform software analysis operations. Metric plug-ins are OSGi services that implement a common interface and are discoverable using the plug-in administrator service. In practice, all metric plug-ins inherit from an abstract implementation of the plug-in interface and only have to provide implementations of 2 methods for each binding datatype (`run()` and `getResult()`) and the `install()` method to register the plug-in to the system.

Each plug-in is associated with a set of activation types. An activation type indicates that a plug-in must be activated in response to a change to the corresponding project asset; this is the name of the database schema entity that models the asset and therefore the metric is activated each time a new entry is added to the database table. A metric plug-in can define several metrics, which are identified by a unique name (*mnemonic*).

Table 4.4: List of metrics included in the SERP default dataset

Plug-in	Data Description	Activator	Metrics
Size	Calculates various project size measurements, such as number of files, lines and documentation for various types of source files.	`ProjectFile` `ProjectVersion`	11
Module	Aggregates size metrics per source code directory and calculates the average source module size per version.	`ProjectFile` `ProjectVersion`	4
Structural	Calculates McCabe's and Halstead's metrics for C/C++ and Java.	`ProjectFile`	13
Maintainability	Calculates the Maintainability index, as defined in [CALO94] per source code module. It also produces an average per project.	`ProjectFile` `ProjectVersion`	2
Discussion Heat	Identifies heated email discussions and evaluates their effect on source code intake.	`ProjectVersion` `MailingListThread`	3
Developer Statistics	Calculates the number of developers that worked on a file ("eyeball" metric) and the number of those that are active in the project at any given time.	`ProjectFile` `ProjectVersion`	5

Each metric is associated with a scope that specifies the set of resources this metric is calculated against: for example files, namespaces, mailing lists or directories. Metrics can also declare dependencies on other metrics, the system will use this information to adjust the plug-in execution order accordingly through the metric activator service. Metric results are stored in the system database either in predefined tables or in plug-in specific tables. The retrieval of results is also bound to the resource state the metric was calculated upon.

To account for the fact that each plug-in might need to store intermediate results or other data in structured format, the extension mechanism was designed to take advantage of the object oriented nature of the database. SERP plug-ins can create and install custom entities that extend the basic metadata schema. The stored data will be persisted to the database, appropriately linked with basic system entities. The custom tables are isolated from the rest of the database, as the runtime types required to access them are restricted to the plug-in.

The SERP platform comes with a default set of 38 metrics, which are described in Table 4.4.

4.4 Operation

The SERP research platform is built by importing projects into the metadata database and running analysis tools on the imported projects. The processing is split in two phases:

Import phase: The system converts raw data from the project's data stores in metadata entities which it stores in the database. The system only performs this operation once for each imported project and can then synchronise the metadata state with the raw data state when the raw data have been updated.

Analysis phase: The analysis tool plug-ins operate on project data and extract measurements, which they store in the database.

One of the key properties that SERP was designed to maintain is data consistency. In a high data volume, heterogeneous multi-processing environment, errors should be treated as a fact of the platform's operation, not as exceptional cases. Errors can come from various sources:

- Non-handled corner cases: When an analysis plug-in is developed, it is usually tested against a small dataset in order to keep testing time low. This type of testing usually cannot account for all cases which appear in data originating from heterogeneous sources.
- Race conditions: Analysis plug-ins that were not designed for multiprocessing might starve the system resources or stale the system when waiting requests that cannot be satisfied.
- Failures in external components: SERP depends of various external software and hardware components that might fail.

The design principle employed was the atomicity of operations. Specifically, all operations that affect the metadata schema are modelled as atomic executable units, which in practice are scheduler jobs (see Section 2.2.3.1) with an attached open database transaction. The database plays an important role, as it is employed by SERP to notify it when a transaction remains active for an excessive amount of time and also to revert to a previous safe point when processing fails. The required database characteristics imply that SERP can only run on databases with ACID (Atomicity, Consistency, Isolation, Durability) [RH83] guarantees and transaction resource management facilities. Moreover, all jobs are required to fail fast; when an error occurs, by convention the job must not attempt to provide a work around. A job failure will be automatically cascaded to all dependent jobs.

As a result of the above, the processing phases are completely independent and isolated. This means that the metadata updater and the various analysis tools can operate on the same project simultaneously as the metadata updates are hidden from the analysis tools until the metadata are in a consistent state.

4.4.1 Representing SCM Data in Relational Format

The import phase is a two stage process. In the first stage, the system extracts metadata from the raw data and inserts them in the database. The depth of data analysis required for storing metadata from each data source varies significantly; BTS metadata are copied verbatim in the database, while SCM metadata must be processed in order to extract a virtual file tree representation from a very poor source of information, the SCM log. In the second phase, the system creates links between different types of metadata or creates different organisations between metadata of a single type. In this section, we describe the algorithms we have contributed to the SQO-OSS tool updater service (Section 2.2.3.1).

The SERP enables applications to access the full SCM repository for a large number of software projects. However, the majority of analysis algorithms and tools do not work directly with SCM logs; they work with file contents and directory structures. Moreover, several analysis methods only work with specific file types (i.e. source code files); acquiring filtered lists of files directly from the SCM for a specific version is not a supported operation in most SCMs. To provide such services to analysis tools, SERP maintains detailed metadata for each project version in the SCM system.

The SCM metadata updater is used to convert SCM data to a relational format. Specifically, it parses and extracts information from the SCM revision log and will convert it to a schema that enables the SERP to retrieve portions of the file tree structure for a specific version instantly. This is an operation equivalent to checking out a specific version from the repository and retrieving a recursive list of the files and directories that comprise it. Converting SCM data from semi-structured to fully structured format has several advantages, in addition to providing the appropriate input to analysis tools:

- A carefully designed schema can accommodate both distributed and centralized SCM metadata, effectively abstracting their semantic differences. Tools that do not rely on SCM-specific data can therefore be applied on any project, independently of its SCM system.

- A Database Management System (DBMS) allows the attachment of other data, such as measurements, to the stored metadata in a cross-SCM manner, while there is currently no SCM system that can do that. Therefore, SERP can support the storage and retrieval of measurements for each file and project version state.

Table 4.5: Description of the input to the scmmap algorithm

Field	Type	Description
Rev	`String`	The SCM's unique revision identifier.
Auth	`String`	The SCM user name for person that performed the commit.
Date	`Long`	The timestamp of the commit. The resolution depends on the underlying SCM, although most support millisecond accurate timestamps.
Branch	`String`	The branch this commit affects (if any).
Tag	`String`	The tag this commit defines (if any).
Msg	`String`	The commit message, as entered by the commiter.
NormalOps	`Map{Path->ChType}`	A structure that maps the paths affected by the commit to the type of operation that was performed on the path.
Path	`String`	The affected path.
ChType	`enum{A,M,D}`	The type of operation that was performed on a path, `A` for additon, `M` for modification, `D` for deletion.
CopyOps	`List{CopyOp}`	A list of copy operation descriptors, whose format is described below.
From	`String`	The copy operation originating path.
FromRev	`String`	The version the copy operation originates from.
To	`String`	The destination of the copy operation.
ToRev	`String`	The destination revision of the copy operation.

- A modern DBMS will use multilevel indexes to speed up search and manipulation of records. Most SCM systems currently scan all project versions up to the requested one to retrieve a versioned resource (except from GIT which uses indexed access to objects). By using a relational mapping, most SCM metadata operations (e.g. log retrieval, version of 3^{rd} modification for a file) can be reduced from $O(n)$-class problems to $O(\log n)$ or $O(1)$ class, thereby accelerating analysis tool execution.

- Filtering and selecting the specific files to retrieve from the SCM system is faster if it is done outside it. For example, if an analysis tool requires a listing of all source code files in a revision, a DBMS will search through the file metadata at least an order of magnitude faster than it would take for an SCM system to checkout the project and then produce the listing.

For the reasons outlined above, in SERP we used an intermediate database schema for storing SCM metadata. The schema has been described in Section 4.2.2. The initial design target for the metadata schema was to support the full set of operations supported by the SVN repository format, and, to the best of our knowledge, this is the first description of the algorithms that can be used to populate it. However, as we show, the schema can also support the GIT repository format with minor modifications. For brevity, we refer to the mapping algorithm as scmmap .

```
svn log -v -r4589
------------------------------------------------------------------------
r4589 | gousiosg | 2009-02-26 18:07:47 +0200 (Fri, 26 Feb 2009) | 2 lines
Changed paths:
   M /trunk/metrics/Makefile
   A /trunk/metrics/discussionheat (from /trunk/metrics/skeleton:4579)
   M /trunk/metrics/discussionheat/pom.xml
   D /trunk/metrics/discussionheat/src/eu/sqooss/impl/metrics/skeleton
   A /trunk/metrics/discussionheat/src/eu/sqooss/metrics/discussionheat
   A /trunk/metrics/discussionheat/[...]//discussionheat/DiscussionHeat.java
   A /trunk/metrics/discussionheat/[...]/discussionheat/DiscussionHeatActivator.java
   D /trunk/metrics/discussionheat/src/eu/sqooss/metrics/skeleton
   M /trunk/metrics/pom.xml

Calculates the "heat" of a discussion by classifying email threads according to
number of emails and max depth
------------------------------------------------------------------------

git log --find-copies-harder --pretty=fuller --stat --summary 5553ec7c657
------------------------------------------------------------------------
commit 5553ec7c657f45cc4c76a9d6172fa687b1db5aee
Author:     gousiosg <gousiosg@gmail.com>
AuthorDate: Thu Feb 26 16:07:47 2009 +0000
Commit:     gousiosg <gousiosg@gmail.com>
CommitDate: Thu Feb 26 16:07:47 2009 +0000

Calculates the "heat" of a discussion by classifying email threads according to
number of emails and max depth

git-svn-id: http://anonsvn.sqo-oss.org/trunk@4589 3fa48db9-4f4b-0410-a8e8-[...]

metrics/Makefile                                   |    2 +-
metrics/{skeleton => discussionheat}/Makefile      |    0
.../manifest/manifest.mf                           |    0
.../manifest/manifest.mf.example                   |    0
metrics/{skeleton => discussionheat}/pom.xml       |   16 ++--
.../metrics/discussionheat/DiscussionHeat.java     |   89 ++++++++++++++++++++++
.../discussionheat/DiscussionHeatActivator.java}   |   22 ++----
metrics/pom.xml                                    |    1 +

8 files changed, 105 insertions(+), 25 deletions(-)
copy metrics/{skeleton => discussionheat}/Makefile (100%)
copy metrics/{skeleton => discussionheat}/manifest/manifest.mf (100%)
copy metrics/{skeleton => discussionheat}/manifest/manifest.mf.example (100%)
copy metrics/{skeleton => discussionheat}/pom.xml (89%)
create mode 100644 .../eu/sqooss/metrics/discussionheat/DiscussionHeat.java
------------------------------------------------------------------------
```

Listing 4: SVN and GIT logs for the same revision of the same repository. The repository format was initially SVN and was converted to GIT without loss of information, using the `git-svn` tool.

Input At its input, the scmmap algorithm expects a full revision log for the whole repository, sorted in ascending revision order (older version first). The algorithm makes three assumptions about the revision log data that might not stand for all SCM systems:

- The repository must be globally versioned and each commit must report all affected files. This assumption excludes repository formats (such as CVS) that version resources at the file level, even though algorithms exist to convert file-level version data to globally versioned data.
- The repository must be able to sort its revision log in an "older first" fashion, even across branches or tags. The definition of "older" is left to the repository.
- If the repository supports inter-repository resource copying, then the repository must be able to maintain and report information about the copy operations.

The scmmap expects at its input an ordered list of tuples, whose format is presented in Table 4.5. Since the main purpose of the algorithm is to convert information about changes in file paths to a relational format, the algorithm input mostly comprises of descriptions of the changes to file paths.

To demonstrate how data from both SVN and GIT can be used as input to the algorithm, we converted the SCM repository we used for the development of SERP from SVN to GIT. We then used each tool to extract the log for the same changeset, which can be seen in Listing 4. We then used extracts of the log files to populate the fields of Table 4.6. The selected commit performs several complicated operations, for example it copies a directory and modifies several files, while also deleting others. The conversion to GIT was performed without loss of information. As we can see, both SVN and GIT can provide adequate information to be used as input to the scmmap algorithm. Differences do exist; for example, a file path in GIT can belong in multiple branches at the same time, as GIT does not attach versions to paths as SVN does. Furthermore, in real deployments, such as the ones the system processes, GIT attaches cryptographic signatures to each changeset, which hold information about the people that have approved the specific changeset before it is committed to the repository. This additional information is not yet used by the scmmap algorithm.

Output The SERP metadata schema is described in Section 4.2.2. The scmmap algorithm primarily fills in the appropriate values in the tables `ProjectVersion` (PV) and `ProjectFile` (PF). It also affects values in tables `Directory` (DIR), `Tag` (TAG) and `Branch` (BRN). The original SERP schema allows the co-existence of data from various projects. In order to simplify the algorithm description, in the following sections we assume that only one project is stored in the schema. This simplification does not affect the algorithm design.

Relational Algebra Notation

The relational algebra we use in algorithm descriptions was introduced by Codd in his classic work on relational databases [Cod70]. In relational algebra, data are organised in tuples, or ordered sets, whose fields $(f_1, \cdots f_n)$ are called attributes. Relations are sets of similar tuples. Relational algebra defines six primitive operations while also borrowing from set theory. In our text, we use the following notation:

Projection $\pi_{f_1,f_2}(R)$ Returns tuples with the values of attributes f_1, and f_2 from R.

Selection $\sigma_{\{R.f_1>val1 \wedge R.f_3=val2\}}(R)$. A selection returns those tuples from a relation that satisfy the selection constraints.

Aggregation ${}_{R.f_1}G_{\max(R.f_2)}(R)$ Applies an aggregate function (one of max, min, avg, count, sum) on attribute f_2 of R in tuples grouped by each different value of f_1. If the group by part is ommited, then the value of the application of the aggregate function on all tuples is returned as a single result.

Join $\begin{pmatrix} R \bowtie S \\ R.f_2 > value \\ S.f_2 = R.f_4 \end{pmatrix}$. A natural join between two relations will return a relation with all columns from both tables populated with tuples which are equal on their common attributes and in addition satisfy the join constraints (expressed below the join specification). Apart from natural joins, we also use semijoins ($R \ltimes S$ or $R \rtimes S$), which are similar to natural joins, except that the returned relation only contains attributes from the first ($\ltimes$) or the second ($\rtimes$) relation.

Set Difference $R \leftarrow R - \{\alpha, \beta, \ldots, \omega\}$ A set difference operation will return a relation with all tuples in R, excluding the one specified at the right hand side of the operation.

Table 4.6: Correspondence of the fields for the SVN and GIT SCM systems to the input fields for the scmmap algorithm. Values correspond to the log messages shown in Listing 4.

SCM Entry Field	SVN	GIT
Rev	`r4589`	`commit 5553ec7c657f[...]5aee`
Auth	`gousiosg`	`gousiosg <gousiosg@gmail.com>`
Date	`2009-01-13 13:54:11`	Thu Feb 26 16:07:47 2009 +0000
Branch	Depends on committed path	`git show 5553ec7c6[...]5aee`
Tag	Depends on committed path	`git show 5553ec7c6[...]5aee`
Msg	The commit message	The commit message
Normal Operations		
Path	`/trunk/metrics/Makefile`	`metrics/Makefile | 2 +-`
ChangeType	`M`, attached to path	`M`, inferred as path is not reported as created or deleted
Copy Operations		
From	`from /trunk/metrics/skeleton`	`metrics/skeleton/Makefile`
FromRev	4579	(implied) immediately previous in branch
To	`/trunk/metrics/discussionheat`	`metrics/discussionheat/Makefile`
ToRev	4589	(implied) current

Basic operations In addition to the relational algebra operations described in the sidebox titled "Relational Algebra Notation" and in order to make the algorithm descriptions simpler, we describe below a set of simple relational algebra operations. The majority of those operate on relations and hence are described using relational algebra notation.

Copy a relation $S \leftarrow \kappa(R)$ Creates a new relation S whose tuples are $1:1$ copies of the tuples in relation R.

Set attribute values $\varepsilon_{(f_1,\cdots,f_n \leftarrow val),\cdots}(R)$ Sets the value of fields $f_1, \cdots, f_n$ to val for all tuples in R. Multiple operations can be combined.

Update attribute values ${}_{\phi}U_{\varepsilon}(R)$: Executes the set attribute value set operations defined by ε on all tuples that match the criteria specified by ϕ. The update operation is semantically equivalent to removing the tuples that match from R, perform the attribute value set operation on the resulting relation and then up performing a union operation between the result of the attribute update and the original relation.

$$
\begin{aligned}
{}_{\phi}U_{\varepsilon}(R) \quad \equiv \quad & S \leftarrow (R - \sigma_{\phi}(R)), \\
& \varepsilon(S), \\
& R \leftarrow ((R - \sigma_{\phi}(R)) \cup S)
\end{aligned}
$$

Relation rename $R\ as\ D$ Temporarily changes the name of a relationship. Used for distinguishing multiple uses of a relationships in a single operation, for example in case of self-joins.

File path functions For a given `path`, `dirname(path)` will return the directory portion while `basename(path)` will return the file name or the directory name portion of `path` depending on whether `path` points to a file or a directory.

Get files in directory $getFiles(D, V)$: A relational operation, semantically equivalent to checking out the directory D in version V and retrieving the (non-recursive) list of files. It is defined as:

$$getFiles(D,V) = \left(DIR \rtimes \left(PF \ltimes \begin{pmatrix} PV \bowtie (PV asPV_2) \bowtie (PV asPV_3) \\ PV2.seq <= PV.seq \\ PV3.seq >= PV.seq \\ PV.id = V.id \end{pmatrix} \right) \right)$$
$$\left(\begin{array}{c} PF.validFrom = PV2.id \\ PF.validUntil = PV3.id \\ PF.state \neq D \\ DIR.path = D.path \end{array} \right)$$

The returned relation comprises of tuples representing all files that are live in the provided directory in the given version. The result may be an empty set.

The scmmap algorithm The algorithm operates on a list of SCM log entries (*Entries*). For each entry, it applies a set of data extraction (`processCommit, processCopyOps, processOps`) and data refinement (`addModifiedDirEntries`, `replayLog`, `updateValidUntil`) operations. It maintains two state variables, a relation for storing the list of entries to the `ProjectFile` table (VF) before they are merged into the database and one to store the currently processed version (Ver). The state variables can be freely manipulated by each algorithm step, but they are cleared after each revision step. We now analyse each algorithm step in detail.

Function *scmmap(Entries)*
Data: *Entries* : An ordered set of SCM log entries
Result: A relational representation of project file metadata
foreach $logentry \in Entries$ **do**
  $VF \leftarrow \emptyset$;
  $Ver =$ processCommit($logentry$) ;
  processCopyOps($logentry$) ;
  processNormalOps($logentry$) ;
  addModifiedDirEntries() ;
  replayLog() ;
  $PF \leftarrow PF \cup VF$;
  updateValidUntil() ;
end

The `processCommit` operation extracts data from the commit log and stores them in the `ProjectVersion` (PV) table. It first determines the correct sequence number for the revision by selecting the maximum sequence number already in the database. It then creates a new entry in the `ProjectVersion` table which also stores in the Ver state variable, to make it available to the steps that follow. If the log entry specifies a tag or a branch, entries are added to the corresponding tables.

Function *processCommit(E)*
$Seq = G_{\max(seq)}(PV)$
$Ver \leftarrow \{E.Rev, E.Date, E.Msg, Seq + 1\}$;
$PV \leftarrow PV \cup Ver$
if $E.Tag \neq null$ **then**
  $T \leftarrow \{Ver, E.tag\}$;
  $TAG \leftarrow TAG \cup T$
end
if $E.Branch \neq null$ **then**
  $B \leftarrow \{Ver, E.branch\}$;
  $BRN \leftarrow BRN \cup B$
end
return Ver

The next step deals with file copies. Copy operations must be processed before normal operations, as some SCM systems allow multiple actions to be recorded in a single commit after a resource has been copied. For example, in SVN a user can copy a directory (effectively duplicating its entries in another repository path) and then proceed to delete a few files in the copied location while also modifying others. All those actions will be recorded in a single changeset. Processing of the copy operations consists simply of calling the *copyPath* function for each individual copy operation.

Function *processCopyOps(E)*
foreach $Op \in E.CopyOps$ **do**
  $Dirname \leftarrow$ `dirname`$\{Op.FromPath\}$;
  $Filename \leftarrow$ `basename`$\{Op.FromPath\}$
  $$FromFile \leftarrow \sigma_{\{PF.name=Filename\}} \begin{pmatrix} PF \ltimes DIR \\ PF.dirid = DIR.id \\ DIR.path = Dirname \end{pmatrix}$$
  $FromVer \leftarrow \sigma_{\{PV.revisionid=Op.Rev\}}(PV)$
  $copyPath(FromVer, FromFile, Ver, Op.to)$
end

The *copyPath* function duplicates the file or files (depending on whether F_p is a file or a directory) residing under path F_p in version F_v and changes their version to T_v and their path of residence to T_p. If the F_p argument denotes a directory, then the copy operation is cascaded to all files and subdirectories included in F_p. All input variables denote tuples in the PF (F_p) and PV (F_v, T_v) relations respectively, except for T_p which is a string representation of a file path. The *copyPath* function creates tuples that appends to the VF state variable, to make them available to the remaining algorithm steps. To keep track of the history of the file, the `copyFrom` field is set equal to the `id` field of the source of the copy.

Function *copyPath*(F_v, F_p, T_v, T_p) ;
$Dirpath \leftarrow$ `dirname`$\{T_p\}$;
$D \leftarrow \sigma_{\{DIR.path=Dirpath\}}(DIR)$;
$R \leftarrow \kappa(\sigma_{\{PF.id=F_p.id\}}(PF))$;
$U_{R.dirid \leftarrow D.id, R.verid \leftarrow T_v.id, R.copyFrom \leftarrow F_p.id, R.status=A}(R)$;
$VF \leftarrow VF \cup R$;
if $\neg F_p.isdir$ **then**
  return
end
$R \leftarrow getFiles(F_p, F_v)$;
foreach $F \in R$ **do**
  $copyPath(F_v, F, T_v, \varepsilon_{\{F.dirid \leftarrow F_p.dirid\}}(F))$;
end

Next, the algorithm proceeds to process normal (non-copying) operations. Normal operations are different from copy operations in that a normal operation can define a file or directory delete. In SVN, when the user deletes a directory then the log will not contain delete entries for the files included in the deleted directory (GIT will include this information in the revision log). Therefore the algorithm must infer all files in the deleted directory in order to mark them as deleted. This task is performed by the

deleteDir function. A special case of directory deletion happens when a directory is deleted in the same version that the directory has been copied. This special case can be easily distinguished from the normal directory deletion case as there is no entry for the current path in the database already. The *deleteCopiedDir* function handles the copied directory deletion task.

Function *processNormalOps(E)*
foreach $Op \in E.NormalOps$ **do**
  $F \leftarrow \{Ver, E.dir, E.name, E.nodetype, Op.ChType, null, Ver, Ver\}$;
  $VF \leftarrow VF \cup F$;
  if $Op.ChType = D \wedge E.isDir$ **then**
    $Prev \leftarrow \sigma_{\{PF.dir=E.dir \wedge PF.name=E.name\}} \begin{pmatrix} PF \ltimes PV \\ PV.seq < Ver.seq \end{pmatrix}$;
    if $Prev = \emptyset$ **then**
      $Prev \leftarrow \sigma_{\{VF.status=A \wedge VF.name=Op.name \wedge VF.isDir\}}(VF)$
      $deleteCopiedDir(Prev, Ver)$
    else
      $deleteDir(Prev, Ver)$;
    end
  end
end

The *deleteDir* function recursively retrieves all files in a directory that is marked for deletion and in turn marks them as deleted.

Function *deleteDir(DF, V)*
$Prev \leftarrow G_{\{\max(PV_1.seq)\}} \begin{pmatrix} PVasPV_1 \rtimes PVasPV_2 \\ PV1.seq < PV2.seq \end{pmatrix}$
$Files \leftarrow getFiles(DF, Prev)$
foreach $F \in Files$ **do**
  if *F.isdir* **then**
    $deleteDir(F, V)$
  end
  $TD \leftarrow \kappa(F)$;
  $U_{\{Ver.id \leftarrow V.id \wedge TD.status=D\}}(TD)$;
  $VF \leftarrow VF \cup TD$
end

Furthermore, SCM changesets can group together both copying and non-copying operations. If a user first copies a directory and then deletes directory entries within the copied path, the *deleteDir* will not be able to infer the deleted files, as those where not recorded yet in the database. For this reason, the *deleteCopiedDir* function

searches for deleted file entries within the files recorded in the currently processed revision, and marks them appropriately.

Function *deleteCopiedDir(DF, V)*
$Files \leftarrow \sigma_{\{VF.dirid=DF.dirid\}}(VF)$
foreach $F \in Files$ **do**
  if *F.isdir* **then**
    $deleteCopiedDir(F, V)$
  end
  $TD \leftarrow \kappa(F)$;
  $U_{\{Ver.id \leftarrow V.id \wedge TD.status=D\}}(TD)$;
  $VF \leftarrow VF \cup TD$
end

After the end of the *processNormalOps* function, the algorithm has finished processing all information that the SCM system has provided to it. The next steps involve resolving semantic relationships between the processed entries. The first such operation creates fake entries for directory paths whose contents have been modified in the current revision. In this step, the algorithm mimics the behaviour of a real file system; when a file is inserted or updated in a directory, most file systems will update the directory's access time, thereby creating a new state for it. The `scmmap` algorithm does the same for the virtual file tree it creates. This enables measurements to be attached to directories using exactly the same storage type (the `ProjectFileMeasurement` entity) as in the case of normal files, thereby simplifying plug-in design.

Function *addModifiedDirEntries()*
$Files \leftarrow \sigma_{\{\neg VF.isdir\}}(VF)$
foreach $F \in Files$ **do**
  $D \leftarrow DIR \ltimes F$;
  $Dpath \leftarrow$ `dirname`$\{ D.path\}$;
  $Dname \leftarrow$ `basename`$\{ D.path\}$;
  $D \leftarrow \sigma_{\{DIR.path=Dpath\}}(DIR)$;
  $DF \leftarrow \sigma_{\{PF.name=Dname \wedge PF.dirid=D.id\}}(PF)$;
  $MD \leftarrow U_{\{DF.state=M\}}(\kappa(DF))$;
  $VF \leftarrow VF \cup MD$
end

At this stage, the `scmmap` algorithm has created all required entries for project file changes in the temporary VF relation. A diligent reader might have already noticed that the VF table might contain duplicate entries; for example the $addModifiedDirEntries$ method will create duplicate entries for the same directory when two files have been modified in this directory. A more subtle situation occurs when a directory has

been copied to a new location and a file has been removed from the new location; the same file will appear initially as added and then as deleted. To resolve such inconsistencies, the scmmap algorithm must work out the prevailing state for each file, and it does so by looping over all instances of the file and applying a point system.

```
Function replayLog() ;
foreach F ∈ VF do
    oldpoints = 0; PREV ← ∅ ;
    foreach P ∈ VF do
        if F.name ≠ P.name ∨ F.dirid ≠ P.dirid then
            next P
        end
        switch P.chType do
            case A
                points ← 2
            endsw
            case M
                points ← 4
            endsw
            case D
                points ← 8
            endsw
        endsw
        if points >= oldpoints then
            oldpoints ← points ;
            if PREV ≠ ∅ then
                VF ← VF − PREV
            end
            PREV ← P
        end
    end
end
```

After the log replay step, all entries that modify the virtual file tree maintained by SERP have been appended to the appropriate relations. In the final step, the algorithm updates the `validUntil` fields for the entries in the `ProjectFile` relation. To do so, it retrieves all files that are live in the immediately previous (as recorded by the version sequence field) version and updates the appropriate field with the *id* of the current version, for those files for which there is no new version that has been processed in the current project version. A new file version existence is determined by searching for each old version file for a corresponding file in the current version set that resides in

the same directory and has the same name.

Function *updateValidUntil()*

$$Prev \leftarrow G_{\{\max(PV_1.seq)\}} \begin{pmatrix} PVasPV_1 \rtimes PVasPV_2 \\ PV1.seq < PV2.seq \end{pmatrix}$$

$$PrevFiles \leftarrow \begin{pmatrix} PF \ltimes \begin{pmatrix} PV \bowtie PVasPV_2 \bowtie PVasPV_3 \\ PV2.seq <= PV.seq \\ PV3.seq >= PV.seq \\ PV.id = Prev.id \end{pmatrix} \\ PF.validFrom = PV2.id \\ PF.validUntil = PV3.id \\ PF.state \neq D \end{pmatrix}$$

foreach $F \in PrevFiles$ **do**

  $N \leftarrow \sigma_{\{VF.name=F.name \wedge VF.dirid=F.dirid\}}(VF)$;

  if $N = \emptyset$ **then**

    ${}_{\{F.id=PF.id\}}U_{\{F.validUntil \leftarrow Ver.id\}}(F)$;

  end

end

After the end of the update process, the SERP system is fully equipped with the appropriate metadata to respond to metadata queries much faster than comparable approaches that involve access to the raw data. The relative volume of the stored metadata is comparable to that of raw data. The metadata enable complex metrics that need to read the contents of project artifacts to benefit from the database's ability to filter out unwanted items before they reach for the data retrieval subsystem; for example a metric interested in a subset of the project files (e.g. all source code files) can request just those and the system will automatically filter out irrelevant entries, thereby saving the time to fully checkout and then clean up a full project revision. The time savings are significant: on our system, a query to retrieve all source code files for version 135332 of the FreeBSD project executes in two seconds. A comparable approach would entail checking out all files from the repository and then selecting the required ones: on our system it takes 14 minutes for the same version. Furthermore, the metadata entities are also used by metrics to store and calculate results in an incremental fashion; for example, when the source code updater encounters a new revision, it will notify all metrics that calculate their results on whole project checkouts and, after the result is calculated, it can be stored against the same database object.

Implementation The algorithm has been implemented as an extension to the SERP metadata update service. The current implementation only supports the SVN repository format as at the time of this writing there is no library in Java that implements

the required set of GIT repository functionality. The implementation itself posed a significant challenge, and the design of the algorithm went through various iterations (the corresponding file received 132 commits, more than any other in the SERP repository) until the storage schema was satisfactory.

In the form presented above, the algorithm will maintain a virtual file tree of all files that are live in each project revision, which also includes files copied in directories denoting branches and tags. As the representation of the file tree in SERP closely resembles the one in SVN, all files will remain live until they are marked as deleted, and this also includes the files marked as branches or tags. As both branches and, especially, tags usually receive a very limited number of updates, maintaining the liveness of files (the last step of the algorithm) can be problematic for large projects. In the implementation of the scmmap algorithm, we give the user the option to exclude certain paths from the processing loop. This complicates the implementation significantly, as there are several cases where a file is copied from a non-processed path to a path that can be processed. In such cases, we resort to the repository to retrieve the missing information about the source file of the copy action.

4.4.2 Resolving Developer Identities Across Data Sources

One of the goals of the SERP platform is to integrate data from various data sources and to enable analysis across those data sources.

Direct links are links across data sources based on items that are common (or similar) across the data sources. This kind of links exists in all projects. An example of common objects is developer identities.

Indirect links result from tagging items in a data source with information specific to other data sources. An example of indirect linking is the inclusion of a bug report number in a commit message that affects this bug report.

Developer identities are *almost* direct links among project data sources. In Table 4.7, we present all different types of identities a developer can have in the data sources that the SERP platform processes, in the ideal case. As all systems managing the processed data sources have been developed independently, there is no guarantee that a single identity will be used across them. Robles [RGB05] categorises the identities in two categories: primary, which are mandatory for accessing the data stores, and secondary, which may not be present, but if they are they provide additional information about the developer. Even though developer identities do exist in all data sources that the SERP processes, these cannot be used without preprocessing. As opposed to the ideal case presented in Table 4.7, in datasets stemming from real OSS projects, developer identities are problematic to process because:

Table 4.7: Examples of identities for the same developer in various data sources.

Source	Identity
SCM (SVN)	`username`
SCM (GIT)	`usename@domain.org`
	`Name Surname <username@domain.org>`
Mailing list	`Name Surname <username@domain.org>`
	`username@domain.org`
BTS	`usename@domain.org`

- A person might have several identities in the course of a project. For example, it is common for developers of long running projects to change their email addresses, for example due to changes in their employment status.

- For most projects, there is no direct mapping between the identity of a developer in the SCM system (which in most cases it is stored as a user name) to the email address used for posting to mailing lists or to submit bugs. Even for projects that maintain central databases of users, this data is usually not available.

- The legal status of developer related data is dubious; although developers sending an email or submitting a patch to a project provide a de facto consent to the project to publish their identities, this consent does not automatically extend to processing the identity or linking it to the other identities the developer may have.

For the purpose of resolving identities across data sources, we designed and implemented the idmap algorithm, a heuristic-based method to identify and group together developer identities for the SERP.

Input The algorithm operates directly on SERP database entities and, more specifically, on entries to the `Developer` and `DeveloperAlias` tables. To account for the average case, we consider a fully resolved, unique identity as a record consisting of a real name, a user name and several emails. We acknowledge that a developer might also have several SCM user names in the lifetime of the project, but based on what is reported by [RGB05], this possibility is very thin.

The `Developer` and `DeveloperAlias` tables are populated with unprocessed data during the metadata update phase. When processing updated raw data versions, metadata updaters link the metadata entity representing the raw data change to an entry in the `Developer` table to mark a change as performed by a certain developer. Depending on the updated raw data, the developer information can be a username or an email possibly followed by the developer's real name. The raw data updaters do not

process the developer data more than checking for duplicate values before appending them in the `Developer` and `DeveloperAlias` table.

To enhance the resolution process, we downloaded the Ohloh developer database through its web Application Programming Interface (API). Ohloh[1] is a web site that analyses the source code evolution of a large number of OSS projects. Similarly to SERP, it uses each project's SCM repository for that purpose; in addition, it allows developers to submit additional information to their profile page, which is what makes the Ohloh dataset particularly useful for our purpose. Several developers have submitted their email and attached it to the user name they use in the SCM they work. Ohloh only distributes emails cryptographically hashed; however, given any email, we can produce its hash and then search Ohloh's database for the user name this email is attached to, thereby providing us with an additional source of information. As the submitted email is what Ohloh users use to register with the service (so it needs to be a real email address), the Ohloh dataset is a premium source of data for the idmap algorithm.

Output The idmap algorithm operates directly on the `Developer` and `DeveloperAlias` tables and reduces the number of identities that the metadata updaters have inserted in those tables, after resolving multiple identity instances into a single one. The idmap algorithm also updates the corresponding raw data metadata to point to the appropriate developer identities after the resolution has been performed. We do not cover this step in the algorithm description.

The idmap algorithm The idmap algorithm is derived from the algorithm presented by Robles in [RGB05]. Robles's approach involved a set of heuristics that were used to fill in a match table, that was later inspected manually for correctness. In our case, the whole process is automated and therefore we had to design the idmap algorithm conservatively, otherwise our sample would be polluted with false positive matches. Additionally, idmap introduces approximate string matching techniques [Nav01] and external datasets (Ohloh) as an additional step to validate the quality of heuristic based matching, which is common between Robles's algorithm and idmap .

The idmap algorithm iterates over a given set of developer identities and for each one it attempts to find a developer whose user name matches exactly or approximatively the real name or the email name of the examined developer. The algorithm uses a scoring system to evaluate the quality of the match; as the name to be matched is tested against various heuristics, a score is updated based on how strong the heuristic is. A match is only considered successful if it scored above a certain threshold and matched against more than one criteria. The criteria being used are analysed in Table 4.8.

[1] `http://www.ohloh.net`

Function *idmap()* ;
M $\leftarrow \emptyset$;
foreach $\{D \in DEV | DEV.name \neq null\ \}$ **do**
  if $OD \leftarrow \sigma_{\texttt{sha1}\{D.email\}=OHLOH.email}(OHLOH)$ **then**
    if $Uname \leftarrow \sigma_{\{DEV.uname=od.uname\}}(DEV)$ **then**
      M $\leftarrow$ M $\cup \{D, Uname, 12\}$
      next
    end
  end
  if $Dev \leftarrow \begin{pmatrix} DEV \ltimes DEVAL \\ DEV.uname = \texttt{extrufe}(D.email) \\ DEV.id = DEVAL.devid \end{pmatrix}$ **then**
    M $\leftarrow$ M $\cup \{D, Dev, 12\}$
    next
  end
  namePerms $\leftarrow$ `nameperm(`$D.name$`)`
  foreach $uname \in namePerms$ **do**
    if $Dev \leftarrow \sigma_{\{DEV.uname=uname\}}(DEV)$ **then**
      M $\leftarrow$ M $\cup \{D, Dev, 9\}$
    end
    if $Dev \leftarrow \begin{pmatrix} DEV \ltimes DEVAL \\ DEV.uname = \texttt{extrufe}(D.email) \\ DEV.id = DEVAL.devid \end{pmatrix}$ **then**
      M $\leftarrow$ M $\cup \{D, Dev, 9\}$
    end
    distMatch $\leftarrow \sigma_{\{\texttt{strdist(}D.uname,uname\texttt{)}<3\}}(DEV)$
    foreach $Dev \in distMatch$ **do**
      M $\leftarrow$ M $\cup \{D, Dev, 3 -$ `strdist(`$Dev.uname, uname$`)`$\}$
    end
    phonMatch $\leftarrow \sigma_{\{\texttt{strdist(phon(}D.uname\texttt{), phon(}uname\texttt{))}<5\}}(DEV)$
    foreach $Dev \in phonMatch$ **do**
      M $\leftarrow$ M $\cup$
      $\{D, Dev, 5 -$ `strdist(phon(`$Dev.uname$`), phon(`$uname$`))`$\}$
    end
  end
  Results $\leftarrow \sigma_{\{R.sum(M.score)>=12\}}((_{\{M.left,M.right\}}G_{sum(M.score)}(M))$ as $R)$;
end
return Results

Algorithm 1: Resolving developer identities — the idmap algorithm

Table 4.8: List of heuristics used by the idmap algorithm

Heuristic	Function	Score	Description
Ohloh	—	12	The examined entry's email is in the Ohloh dataset. As Ohloh distributes emails cryptographically hashed, the `sha1` digest function must be applied to the original email before comparison.
Email username	`extrufe`	12	Use the examined entry's email username (the part of the email before the `@` symbol) to match against entries with the same username.
Name Permutations	`nameperm`	9	Use letters from the examined entry's real name to produce a set of ten possible usernames, then match those against other entries' usernames. For example for the name `John M. Doe`, the heuristic will match entries with usernames like `jdoe, doej` and `jmd`.
String distance	`strdist`	3 - *dist*	Calculate the Levensthein distance [Lev66] between the examined developer's username and other user names.
Phonetic	`phon`	5 - *dist*	Produce a phonetic code (with the double Metaphone algorithm [Phi00]) for each name permutation for the examined developer, calculate string distance from phonetic codes of other developer user names.

The idmap algorithm is shown in Algorithm 1. We use relational algebra notation to describe operations performed on database entities, namely DEV (for `Developer`), $DEVALS$ (for `DeveloperAliases`) and $OHLOH$ (for `OhlohDeveloper`). All entities correspond to the SERP database schema described in Section 4.2.2. The algorithm description makes use of several shortcut functions that denote the heuristic in use. They are described in Table 4.8. When a match is found, the idmap algorithm creates an entry holding the two matched identities and the match score into the M temporary relation.

The idmap algorithm operates on developer identities which already have a real name attached, as according to our experiments, the real name provides the best heuristic matches. Moreover, if the real name is set for an identity, then the email field will also be set, as they are both filled in by the mailing list updater, which in turn reduces the comparisons that must be made in order to derive a full identity. The algorithm starts by querying the $OHLOH$ database using the identity's email to retrieve a user name from the $OHLOH$ data source. It then proceeds to extract the user name part from the identity's email and search for that user name in the identities table. This heuristic matches identities with consistently used user names across all data sources, which is often the case for long time project contributors.

The algorithm then creates a list of possible user names by mingling initial letters with parts of the identity's real name, as it is a common pattern (especially in environments that follow the Unix tradition) to use parts of a person's real name or her

initials as a user name. For each of the potential usernames, 4 checks are performed; first, the candidate user name is checked against other user names of those extracted from emails for potential full matches. It is then again checked against all user names using partial string matching algorithms like string distance and phonetic matching.

At the end of the process, the algorithm has aggregated in relation M all matches along with their scores. All the algorithm has to do then is to aggregate the score for duplicate pairs and return those pairs whose aggregate score surpasses a certain threshold. Using the scores we presented in Table 4.8, we set the threshold to 12, as we wanted to ensure that either a full match or a partial match with more than 2 matching criteria was applied before a pair of identities were identified as matching. Obvisouly, the lower the threshold, the more false positive matches the algorithm will return.

Implementation The core idmap algorithm is implemented as a second stage updater in the SERP. Tha algorithmic complexity of idmap is $O(n^2)$. For each identity that is examined, there are at least 6 database queries that must be performed. For a large number of identities, the idmap algorithm can be very slow, if the implementation matches exactly the description provided here. For this reason, the implementation is split; the full match step is performed when the identities are initially stored in the database, during metadata updates. The second phase (approximate matching) is initiated after the metadata import process. That way, a number of identities that can be resolved using the full matching step are not examined by the approximate matching step. Also, the implementation is careful not to perform database queries more than once, by caching intermediate results in memory. This way, the algorithm can examine ten thousand identities in less than 10 minutes.

4.4.3 Clustering

To cope with large data volumes, large SERP installations must use more than one processing nodes. However, running SERP on a cluster is not simple: each cluster node must have access to both the raw data mirrors and the metadata database, while the system should implement mutual exclusion mechanisms to prevent two cluster nodes from processing the same data. On the other hand, the system should automatically manage cluster resources in order to be able to load each cluster node according to its processing capabilities.

Most analysis methods in software engineering evaluate data within the scope of a single project. Typically, analysis methods calculate a result by applying an analysis algorithm on a project state; for example, this is the case with most source code metrics. In other analysis methods however, analysis results for a specific project state can be the result of combining the analysis results of former project states or those of combining

the results of other analysis methods applied on the same project state. To cope with these scenaria, a cluster must be able to manage the analysis dependencies across cluster nodes, which can lead to significant design and implementation complications. Fortunately, the cluster design can be made significantly simpler by noticing that the tasks to be processed by an SERP cluster are *embarrassingly parallel* if a project is constrained on a single processing node. This means that if a cluster needs to process data originating from more than one projects, which is indeed the typical case when building a cluster, then adequate isolation can be achieved by enforcing a mechanism that constrains the processing of all project data on a single node. This way distributed dependency resolution is not necessary and cluster efficiency only depends on load balancing techniques.

SERP is designed to work in heterogeneous cluster environments, where nodes can have different processing capabilities. All cluster nodes share the metadata database and the raw data mirror, the later over a network filesystem. The cluster is headed by the cluster master, a special node whose job is to maintain information about the cluster nodes and to distribute the processing load on them, according to their capabilities. Processing nodes run a special service which is used by the cluster master to schedule work items and to discover the node's status. The functions offered by the cluster service are described in Table 4.9.

During cluster initialisation, the cluster master discovers the nodes through their entries in the shared database. It then proceeds to retrieve the configuration parameters for all nodes and builds a cache for those. In regular intervals, the master will query each cluster node for runtime parameters. The nodes are split in metric job processing nodes and metadata update nodes, the later being the most powerful. The master decides which node to assign a set of jobs, which are instantiated by the metadata updater (raw data) or on user request (metric plug-ins), based on a scheduling algorithm.

The proposed cluster scheduling algorithm is presented in Algorithm 2. Initially, the algorithm attempts to build a set of scheduling candidate nodes, by excluding those nodes that are already under load. From the list of suitable nodes, the algorithm will return the one that will flush its work queue the fastest, based on the rate of job processing and the remaining queue length variables. If no job processing node is available, then it will search for suitable idle nodes among those that are marked for metadata updates. Metadata update jobs have higher priority over metric run jobs so the job scheduler will execute them first should they appear on the work queue. From the metadata update nodes, the algorithm will select the one with the least system load. In case there is no processing node available for running the update job, the system will save the update request and will reprocess it later.

In its current implementation, the SERP cluster service is much simpler than the

Table 4.9: Cluster services supported commands, results and scheduling implications

Command	Acro	Result
Configurable Parameters		
`getMaxQueueLength`	MQL	The maximum work queue length. Depends on available memory on the node.
`getNumProcessors`	NP	Number of processor equivalents. Not the real number of processors, but an approximation of how many times faster a node is in comparison with a 2GHz processor.
`getSupMDUpdates`	SMD	Whether or not the current node supports metadata updates. As metadata synchronisations are very resource intensive, only the fastest of nodes should run those.
Runtime Parameters		
`getCurQueueLength`	CQL	The number of jobs on the queue. If more than 75% full, or if the number of jobs to be scheduled exceeds the max length, the master will not schedule new jobs.
`getSysLoad`	SL	The load, as reported by the operating system. If more than the reported processor equivalents, the cluster master will not schedule jobs on this node.
`rate`	R	Number of jobs finished per second. Only calculated when jobs are queued.
Operations		
`run`	RUN	Accepts XML descriptions of jobs to be scheduled. The job id is returned if the job was enqueued successfully.
`getFinishedItems`	FNS	Retrieves the job id's for finished jobs.

Function *schedule(N, L)*
Data: N : Set of cluster nodes, L : Number of jobs to schedule
Result: A node to schedule a list of jobs on.
$C = \emptyset$;
foreach $n \in \sigma_{\neg SMD}(N)$ **do**
  if $NP_n > SL_n$ **then**
    next n
  end
  if $CQL_n > 0,75 * MQL_n$ **then**
    next n
  end
  if $L > (MQL_n - CQL_n)$ **then**
    next n
  end
  $C \leftarrow C \cup n$
end
if $C = \emptyset$ **then**
  $C \leftarrow \sigma_{SMD \wedge ((MQL - CQL) < L)}(N)$;
  if $C = \emptyset$ **then**
    sleep ;
    return *schedule(N, L)*
  end
  return $\sigma_{min(SL)}(C)$
else
  return $\sigma_{min(R*CQL) \wedge \neg SMD}(C)$;
end

Algorithm 2: Cluster Job Scheduling Function

design presented here. Currently, it will not assign projects to cluster nodes automatically, nor will it delegate metric runs to the appropriate node. However, it does restrict projects on specific cluster nodes and differentiates jobs between metric updates, which occur on cluster nodes, and metadata updates which run on the cluster master node.

4.5 Summary

In this chapter, we introduced the SERP and described its basic constituents. We also described in depth our contributions to the SQO-OSS tool. We have formally described the first, to the best of our knoweledge, algorithm that can map distributed and centralised SCM repository information on a relational schema and also presented novel solutions to the problems of identifying developer identities across data sources and scheduling jobs on an SERP cluster.

Chapter 5

Empirical Validation

The true worth of an experimenter consists in his pursuing not only what he seeks in his experiment, but also what he did not seek.

— *Claude Bernard*

In the previous chapter, we described the design and analysed the novel aspects of the implementation of a platform that facilitates software engineering research on very large datasets. In this chapter, we validate our design by presenting the excecution of two large case studies that use it. Apart from exemplifying the value of the platform, the case studies produce interesting results, as they provide evidence against two popular conjectures in OSS development: that heated discussions on mailing lists affect the project and that the more developers on a module or a project will make the module's quality better.

5.1 Intense Electronic Discussions and Software Evolution

A well known and well studied (both from a social [Der94] and technical [Coa04] perspective) phenomenon in electronic communications is heated discussions, usually refereed to as "flame wars". Flame wars mainly happen on mailing lists but can also occur on IRC channels or other forms of instant electronic communications. During a flame war the electronic communication etiquette collapses: the topic of discussion stirs away from software development, the rate of message exchanges increases sharply, and the exchanged messages rather than debating the technical aspects of the argument often target directly other participants. Intense discussions do not necessarily signify a flame war; a technical or managerial issue might also elevate the discussion's rate of

exchanges. When the point being pondered is insignificant the discussion is often referred to as a "bikeshed argument" [Fog05], after Parkinson's Law of Triviality [Par58]. Heated discussions are believed to affect a community's social structure; what we try to investigate is whether they also affect the project's evolution in the short term.

5.1.1 Research Questions

Before proceeding to the formulation of the research questions, we must define what can be considered as an intense discussion and how we measure short term evolution.

Let's examine the typical mailing list communication scenario: a discussion starts by a post to a mailing list. When a reply to the specific post arrives, then the discussion is considered open: the first poster might come back with a new question or exit the discussion, while a new poster might join the discussion with a reply to the original message or to the previous reply. A thread might can have multiple branches, when the discussion stirs away from the main topic, usually due to a provocative answer, and consequently multiple exit points. Generally, identifying the message that closes a discussion is very difficult to do automatically for threaded discussions, as it requires analysing the message contents. For this reason, the last email in order of arrival is typically considered as the one that closes a thread.

The diagrams in Figure 5.1 present an overview of mailing list activity, with combined data for all mailing lists in the SERP database. From observing Figure 5.1 we can see that most mailing list threads are less than 5 levels deep and contain less than seven messages. Moreover, most discussions last less than a day. These observations tell us that in order to identify heated discussions without examining the discussion content, we could search for single threads that last less than a day and whose depth level and number of messages lay on the top of both scales. Therefore, if we obtain a list of threads sorted by the number of messages and one sorted by the maximum depth, we could assume that the threads that belong to the top quartile of both lists could be classified as intense discussions. However, this assumption can only stand in the context of a project; as Figure 5.2 shows, different projects, and even different mailing lists in projects, exhibit different list usage patterns, and therefore they must be analysed individually. Regarding mailing list activity, the hypotheses we test in this case study are the following:

H1. The number of messages and the thread depth are dependent characteristics of mailing list activity.

H2. Intense discussion threads can be identified by considering the threads that belong to the top quartile of either the message number or the thread depth distributions for each mailing list.

Once an intense discussion has started, according to popular conjecture, one should

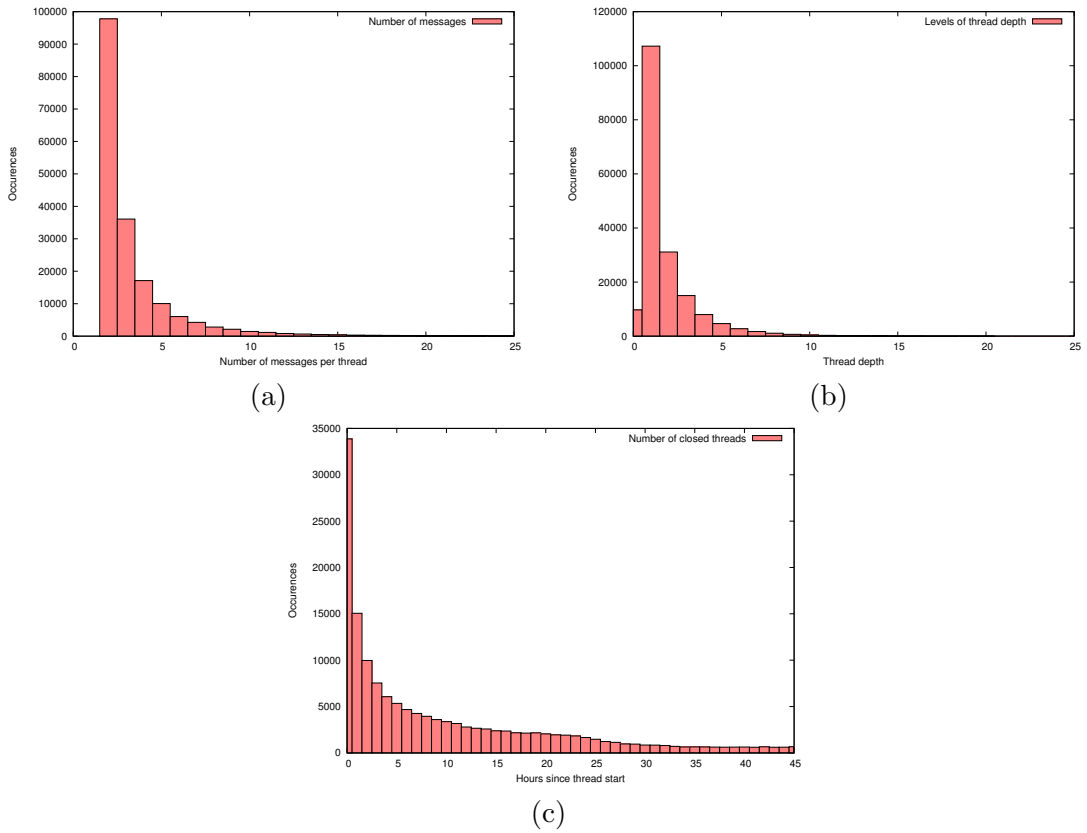

Figure 5.1: Number of messages per thread (a), thread depth (b), and thread duration distributions for all mailing list threads in the SERP database

expect a fall in the average number of lines that are committed to the project's repository, as the developers would be engaged in the discussion. Therefore, the formulated hypothesis is:

H3. During and shortly after an intense discussion, the source code line intake rate diminishes.

To measure the increase or decrease of source code line intake during an intense discussion, we divide time in two periods: if t is the exact time of the first email that provoked the discussion, then $t - 30$ days denotes the *pre-heat* time and $t + 1$ day denotes the *post-heat* time. We can relatively easily retrieve the number of lines that were committed to the repository for both periods and thus derive the number of lines committed per hour. That way, we can assess the effect of an intense discussion on a project's short term line intake by subtracting the two calculated rates.

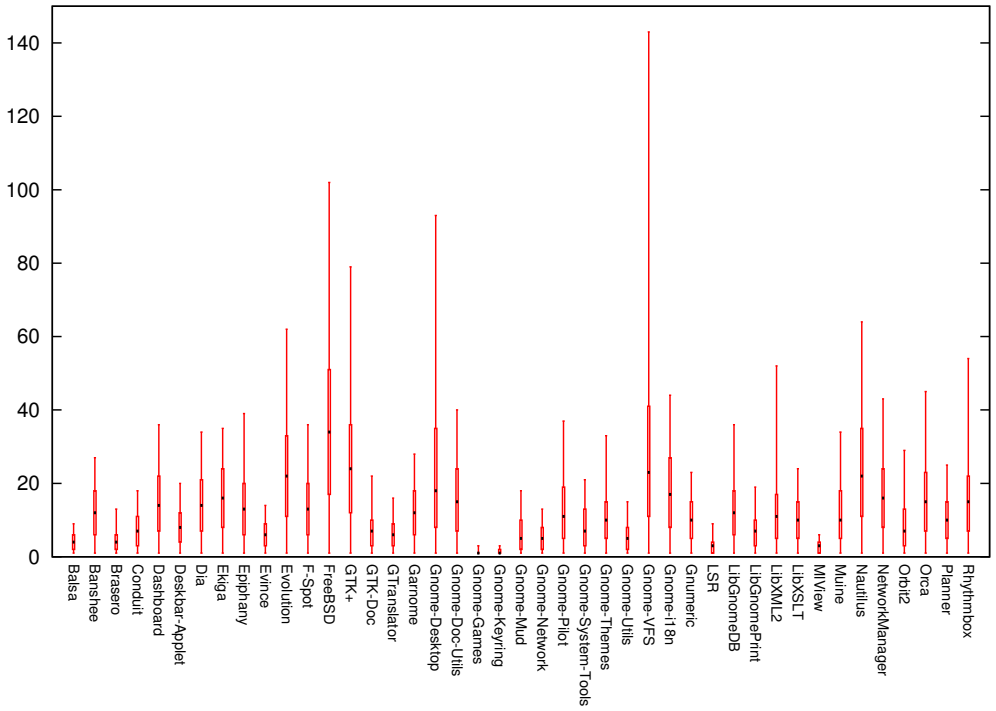

Figure 5.2: Distribution of the number of emails per thread in various projects

5.1.2 Method of Study

To create a plug-in for SERP, a researcher must first identify the potential experiment variables and their relationships with the entities that SERP maintains in its database. SERP calculates metrics incrementally; for each identified resource change, all metrics that are bound to the resource type are calculated for the changed instances of the resource. The analysis tool we describe will need to calculate the depth and the number of emails of a thread, so naturally it will be bound to the `MailingListThread` entity. Each time a `MailingListThread` is updated (i.e. when new emails arrive) the plug-in will be called; to avoid recalculation of threads that have already been evaluated, we will need to cache the result in the database. This can be done by defining a new metric.

The calculation of the rate of line changes for the period before the start of the heated discussion and immediately after it is trickier, as SERP does not currently have any means to store the results of a measurement against a set of entity states. There are two workarounds to this issue:

- store the result against all entity states in the required period, or
- synthesize the result on request, by incrementally combining individual state

measurements.

It is usually cheaper, more precise and future-proof to follow the second route. SERP plug-ins are designed for re-use; if a basic plug-in exports individual measurements of small parts of each assessed resource state, then a high-level plug-in can combine measurements from various other plug-ins to a greater effect. In our case, we designed the described plug-in to calculate the number of lines that have changed in each new project version (and therefore be bound to the `ProjectVersion` entity) and then to use those measurements to calculate the rate of changed lines when a heated discussion is discovered. As expected, the measurement of the lines of code for the changed files is not performed inside our plug-in, but instead we re-use the size plug-in to obtain them for each changed file, as shown in the following code extract:

```
int getLOCResult(ProjectFile pf) {
  AlitheiaPlugin plugin = AlitheiaCore
      .getInstance()
      .getPluginAdmin()
      .getImplementingPlugin("Wc.loc");
  List<Metric> metrics = new ArrayList<Metric>();
  metrics.add(Metric.getMetricByMnemonic("Wc.loc"));
  Result r = plugin.getResult(pf, metrics);
  return r.getRow(0).getCol(0).getInteger();
}
```

Overall, the plug-in is bound to two entities (or activators) and defines three metrics:

VERLOC Stores the number of lines of all text based files that changed per revision.

HOTNESS Attaches a score (ranging from 1 to 4) to a thread based on how "hot" it is. The classification algorithm calculates the number of messages and the thread depth of the message and compares them to the averages of the mailing list that the thread belongs to. A score of 1 to 4 is assigned to each comparison based on the quartile each thread property belongs to.

HOTEFFECT Calculates the effect of heated thread discussions by subtracting the rate of lines of code committed to the repository the month before and the day after start of the discussion.

After designing the plug-in outline and specifying the required metrics, the implementation itself is easy given the wealth of features offered by the SERP API. Metric declarations take a single line of code in the `install()` method:

```
super.addSupportedMetrics("Locs changed in version",
    "VERLOC", MetricType.Type.PROJECT_WIDE);
```

Returning a result stored in the standard SERP schema is equally straightforward. The following code is the actuall implementation of the `getResult()` method for the `ProjectVersion` activator.

```
public List<ResultEntry> getResult(ProjectVersion pv,
    Metric m) {
        return getResult(pv, m,
          ResultEntry.MIME_TYPE_TYPE_INTEGER);
}
```

The following extract is from the `run()` method implementation for the `ProjectVersion` activator. It demonstrates how object relational mapping used in SERP simplifies access to project entities. The code calculates the total number of lines changed in a specific project version and then stores the result to the database.

```
for (ProjectFile pf : pv.getVersionFiles()) {
  if (pf.isDeleted()) {
    linesChanged += getLOCResult(pf.getPrevVer());
  } else if (pf.isAdded()) {
    linesChanged += getLOCResult(pf);
  } else { // MODIFIED or REPLACED
     linesChanged += Math.abs(
         getLOCResult(pf)
       - getLOCResult(pf.getPrevVer());
  }
}
ProjectVersionMeasurement pvm =
   new ProjectVersionMeasurement(m, pv,linesChanged);

dbs.addRecord(pvm);
```

The discussion heat plug-in implementation in total consists of 270 lines of Java code.

We run the plug-in on 64 projects from the Gnome ecosystem and on the full mailing list archives of the FreeBSD project. We did not use all the projects that exist in our platform, as we do not currently have the full email archives for them. For each project, we imported the full source code repository and mailing list archives (current in January 2009) in our SERP installation. In total, the system's metadata updaters processed 409026 revisions (summing up to 2572014 file changes) and 206 mailing lists

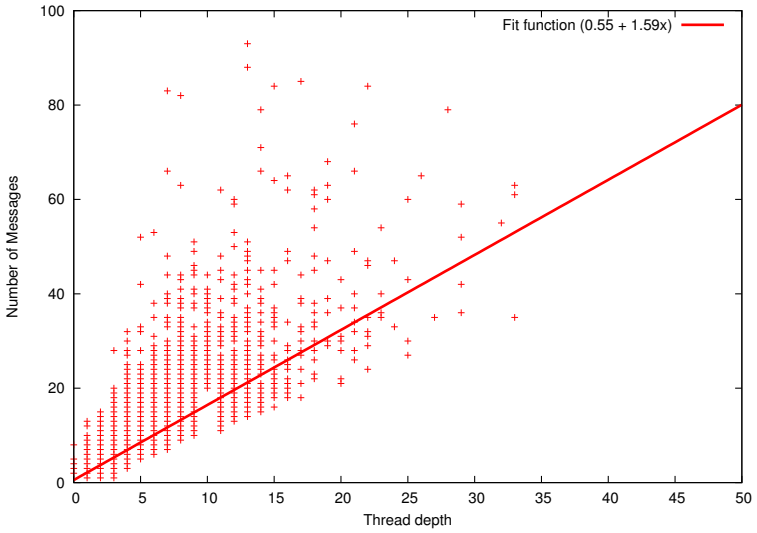

Figure 5.3: Scatter plot of the number of messages vs the thread depth. The two variables are correlated ($R^2 = 0.70$).

with 1134828 email messages organized around 679427 threads. The combined dataset size was more than 40GB.

The discussion heat plug-in workload is mostly database bound; to evaluate its scalability we run the plug-in on a machine featuring a 16-thread 1GHz SPARC CPU and 8GB of memory. The machine is underpowered by modern standards for sequential workloads but provides a very good benchmark for application scalability. We configured SERP to run 24 parallel jobs to compensate for the extensive I/O caused by frequent accesses to the database. During execution, the processor utilization was steadily more than 80%. The database was run on our dedicated 8 core database server on which the CPU load never exceeded 50%. The total execution time was about 6 hours.

5.1.3 Results

After importing the mailing lists to SERP, we calculated the correlation between the number of messages per thread and the average thread depth. Figure 5.3 presents a scatter plot for the two variables. The Pearson correlation coefficient for a linear regression model was calculated at 0.7, thereby indicating a strong correlation between the variables. The corresponding best fit linear model was $N = 0.55 + 1.59 \times D$ where N is the number of messages per thread and D is the thread depth. The correlation between the two variables allow us to validate our first hypothesis. However, since the correlation is not definite, we cannot reject either variable when trying to characterise mailing list behaviour. Thereby, our first hypothesis can be accepted in principle.

Table 5.1: Results from the discussion heat plugin.

Project	**Cases**	**Avg.** HOTEFFECT
Avogadro	1	622
Balsa	1	31
Banshee	3	–1121
Deskbar-Applet	1	–103
Ekiga	2	124
Evince	1	770
Evolution	1	2921
EyeOfGNOME	1	457
FreeBSD	67	–34
Gnome-Desktop	2	2
Gnome-Network	1	106
Gnome-Pilot	1	164
Gnome-Power-Manager	2	–773
Gnome-Themes	1	–2
Gnome-Utils	1	–263
Gnome-VFS	2	33
Gnumeric	1	–230
GTK+	3	183
GTranslator	3	–323
LibXML2	1	187
LSR	1	1150
Meld	1	27
Nautilus	1	1219
NetworkManager	2	124
Orca	3	8
Planner	1	0
Rhythmbox	1	262
Sabayon	1	–244
Sawfish	1	821
Tracker	2	65
Vala	3	–356

Based on the above findings, we define a heated discussion as a mailing list thread that within 24 hours of the arrival time of the email that initiated it, the number of replies are classified to the top quartile of the number of emails per thread variable for the specific mailing list and at the same time the thread depth is above the mailing list median or vice versa. More formally, if n is the maximum value of the number of messages per thread variable and d is the maximum value of the thread depth variable for a specific list, then a thread T is a heated discussion when the following heuristic is true (N_T and D_T are the number of messages and depth for the examined thread T):

$$IsHeated_T = (N_T > 0.75 \times n \wedge D_T > 0.5 \times d) \vee (D_T > 0.75 \times d \wedge N_T > 0.5 \times n)$$

The heuristic maps directly to the results of the HOTNESS metric.

The very strict cut-off threshold employed by the heuristic allowed it to identify just 99 threads (from a total 679427) as intense discussions. We examined 20 randomly chosen threads from those labeled as intense discussions, to validate the heuristic's accuracy. All of them could indeed be classified as intense by a human examiner (pending external validation) as the rate of incoming emails was indeed very high. However, we were unable to find a discussion that could be classified as a flamewar, as most of them were normal discussions around solving technical issues, making decisions or starting or deleting new development branches. This finding suggests that our proposed heuristic is accurate for identifying discussions that go in depth on technical issues but differentiating between flamewars and intense discussions, it needs to be augmented with additional information. Therefore, our second hypothesis is validated.

We then proceeded to examine the effect of intense discussions on the short term evolution of the projects. The results of the HOTEFFECT metric for all identified intense discussions are presented grouped by project in Table 5.1. Overall, we see that there is no significant trend in the change of lines of code intake across projects. Consequently, our third hypothesis must be rejected.

5.2 Development Teams and Maintainability

Software maintenance is a very important part of the software life cycle. It is widely reported that maintenance operations can take up to 80% of the total costs of the software. Consequently, maintainability is one of the top-level criteria in the ISO software quality model and its assessment has been the topic of study of numerous works. Methods for evaluating the effort required for software maintainance include post-effort approaches, such as the mean time to fix a bug, source code analysis techniques and models based on software metrics, that can be used to predict the effort.

As software development is primarily a human oriented task, one should expect that development teams play a significant role in affecting maintainability. Intuitively, one might expect that teams staffed with highly skilled individuals will produce higher quality and, consequently, more maintainable code. A question that arises is whether peer-pressure among colleagues can affect maintainability. For example, it is believed that the OSS development model leads to components of better quality, as the constant monitoring of source code by peers motivates developers to be more careful. [1] Does the same apply on software maintainability? Is the size of the team working on the same project, and the consequent applied peer pressure, an enabling factor for producing maintainable software?

In this case study, we explore the effect of team size on maintainability, as it is

[1] As expressed by Linus Torvalds, creator of the Linux kernel, in the so-called "Linus's Law": *Given enough eyeballs, all bugs are shallow.*

measured and reported by the widely adopted [OH94, WO95, SEI04], but also criticised, MI metric (see Section 2.2.1.1), a composite metric that attempts to quantify maintainability based on low level measurements.

5.2.1 Research Questions

The objective of this work is to study the effects of team size on software maintainability. As teams working on a project or a project module grow larger, one might expect that maintainability will increase, as developers will more eagerly engage into corrective maintenance operations on their code to help their colleagues work with it. The team effect on maintainability might also have a time dimension; to maintainability, it may not be important whether the number of people working on a module increases, but experience suggests that the rate of increase in team membership might play a role. Sharp increases in team sizes can affect project execution [Bro82]; do they play a role in maintainability? We formulate our testable hypotheses as follows:

H1. Team size affects maintainability at the project level.

H2. Team size affects maintainability at the source module level.

5.2.2 Method of Study

To study our hypotheses, we used the SERP. As seen in the previous case study, SERP enables the researcher to write metrics that combine the results of other metrics on request. The MI being a composite metric, it was an ideal candidate for testing and improving the SQO-OSS tool composition and metric execution facilities. Moreover, since SERP already parses and stores the full SCM log to the database, obtaining the number of developers at any moment in the project's lifetime is trivial.[2]

The MI uses several metrics in its formula, namely Halstead's Volume, McCabe's Extended Cyclomatic Complexity, the number of lines of code and number of lines of comments (Section 2.2.1.1). Additionally, the suggested method of calculation is on source code directories [WO95]. Before implementing the maintainability index metric, we implemented the structural metrics plug-in. It calculates an array of code structure metrics, currently for C and Java, which work by partially parsing the source code with a regular expression based parser. The structural metrics plug-in is calculated on a per-file change basis, so its results are bound to the `ProjectFile` entity. The MI plug-in itself is bound on both the `ProjectFile` and the `ProjectVersion` entities as it is calculated both at the module and the version level. To distinguish normal modules from source code modules, it obtains the corresponding information from the module metrics plug-in, in the form of a boolean metric. Also, the MI plug-in uses the size

[2] We have already presented an overview of the technicalities of implementing SERP plug-ins in the previous section, so we will skip detailed technical descriptions of the plug-in internals in this one.

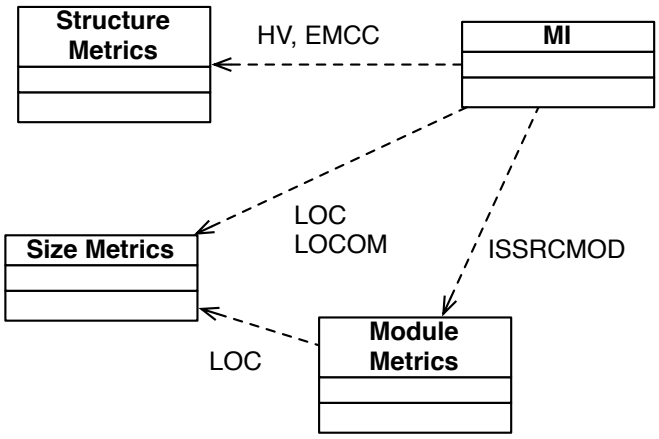

Figure 5.4: Maintainability index plug-in dependencies on other plug-ins.

metrics plug-in to obtain measurements for the lines of code and the lines of comments in each file it processes. A schematic representation of the dependencies between the MI and other plug-ins is presented in Figure 5.4. A sample output of the metric run at the project level for the lifetime of three popular OSS projects can be seen in Figure 5.5.

To calculate the number of developers per resource, we also implemented a new plug-in that queries the database for the number of developers that were active in a time window of one, three and six months. As active, we considered all developers that committed to the SCM repository at least once in each corresponding time window. Constructing a new plug-in was not strictly required, as the measurement to be stored is relatively simple, but since the number of developers per resource type is a common metric for various analyses, implementing such a plug-in can save precious processing and re-implementation time from future plug-ins. Moreover, the corresponding database query proved to be non-trivial for very large projects; for example in the case of FreeBSD it took more than 20 seconds to calculate the measurements for each project version.

All plug-ins were run against the full set of projects that are hosted in the SERP. However, as the structure metrics plug-in was designed to extract measurements only from Java and C files, the MI plug-in only produced results for projects and source modules with more than one C or Java file. As C is commonly used as systems interface language and employed for performance critical algorithm implementations by many programs that are written in higher level languages, there were several cases where a program's primary language was neither C nor Java, but the MI metric had calculated measurements. We filtered out projects whose primary language was not C or Java, by retrieving a list of files from the program's latest version sorted by file type frequency. Most projects filtered were written in Perl, Python or Ruby. After cleaning up the

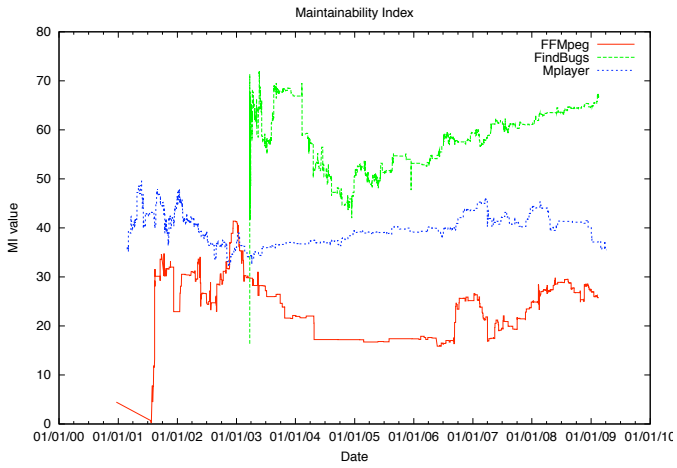

Figure 5.5: Sample maintainability index plot for the whole lifetime of three popular OSS projects.

Language	Module Level	Project Level		
		1 month	3 months	6 months
C	0.007	0.0040	0.0058	0.01264
Java	0.017	0.0472	0.0505	0.0485
Combined	0.02	0.0040	0.0093	0.0127

Table 5.2: MI - Developer count correlation coefficients.

project sample, we were left with 213 projects.

We run all the described metrics on the full dataset on a four machine cluster containing in total of 12 2GHz class CPUs, 16 1GHz class CPUs and a total of 34 GB of memory. The database was run on the most powerful of the machines, but shared CPU power with an instance of SERP running on the same host. Running the structural metrics plug-in was very fast and proceeded at an average rate of 90 jobs per second. Similarly, the MI plug-in whose performance mostly relies on database querying speed and executes simple, index based queries calculated results at the rate of more than 120 jobs per second. The major performance bottleneck proved to be the what we thought of as the simplest of all plug-ins, the developer counting metric. The metric internally uses a series of range queries, which do not scale very well on large datasets. For small projects, it would execute instantly; for larger projects, it could take up to 20 seconds per project revision or file revision. In retrospect, we think that it may have been faster for the system as a whole if we moved the filtering and counting of entries to each cluster node rather than requesting from the DBMS to do the filtering.

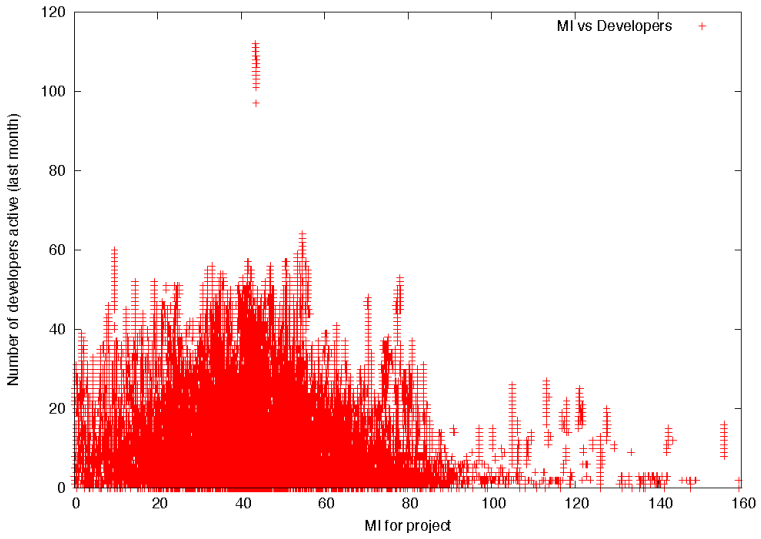

Figure 5.6: MI vs Number of Developers at the project level, for all project versions.

5.2.3 Results

To validate our hypothesis, we extracted and correlated measurements from the MI and developer statistics plug-ins and analysed them statistically. We analysed separately results from Java and C programs at the module and project level to isolate any effects each programming language may have on the results. We also considered different definitions of developer activity, measuring the number of developers that performed a commit in time windows ranging from one to six months. The results can be seen in Table 5.5. In Figures 5.6 and 5.7, we present scatter plots of the MI rating versus the number of developers for module and project level measurements for the combined Java and C results.

In a nutshell, we did not find any significant correlations, so both our hypotheses must be rejected. This result suggests that the number of people working on a project does not seem to affect an important aspect of software quality, maintainability, as measured by the MI metric. Consequently, maintainability may be driven by other factors, such as developer competence or adherence to project development guidelines. New studies need to be performed in order to evaluate the effect of those on maintainability.

5.3 The Perils of Working with Small Datasets

Throughout this dissertation, we stressed the importance of running experiments on very large data samples and in fact this was one of the driving factors that motivated

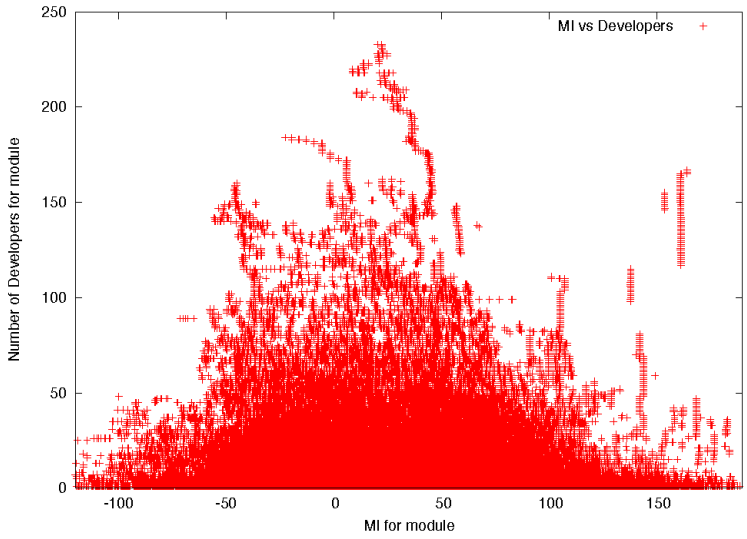

Figure 5.7: MI vs Number of Developers at the module level, for all directories.

our work on SERP. Our experimentation with both case studies revealed interesting side results that, in our opinion, strengthens our expressed thesis by providing motivating examples.

In the first case study, we have rejected our third hypothesis, on the effects of intense discussions on source code line intake, based on the findings presented in Table 5.1. However, a second reading of the same table reveals that we might have equally accepted the hypothesis. If we chose, by random or biased selection, to include in our study just a few very big projects, for example FreeBSD, Banshee and Sabayon, then the evidence would force us to validate our hypothesis. Choosing a few big projects to run experiments on appears to be standard practice, as we have shown in Section 2.3. How can we ensure that the quality of a study is reflectd by the size of the projects it is executed upon?

Moreover, Figure 5.8 presents the distribution of the Pearson correlation co-efficient, as derived by correlating the number of developers per module to the MI for this module on a per project basis. As we can see, there are projects were the two variables are fairly strongly correlated. However, if we examine the sample more carefully, we see that projects that feature a high correlation between the examined variables are small in size or short in history or both, i.e. their data may not be representative of the properties we would like to study. Outliers include projects from the Gnome ecosystem such as Gnome-VFS, Gftp, Metacity and others such as the Exuberant `ctags` project. In our case, we could have used the above mentioned projects and this would still be a valid, scientifically correct, research effort but with completely different results.

There are several issues regarding experimentation methods that emerge from such

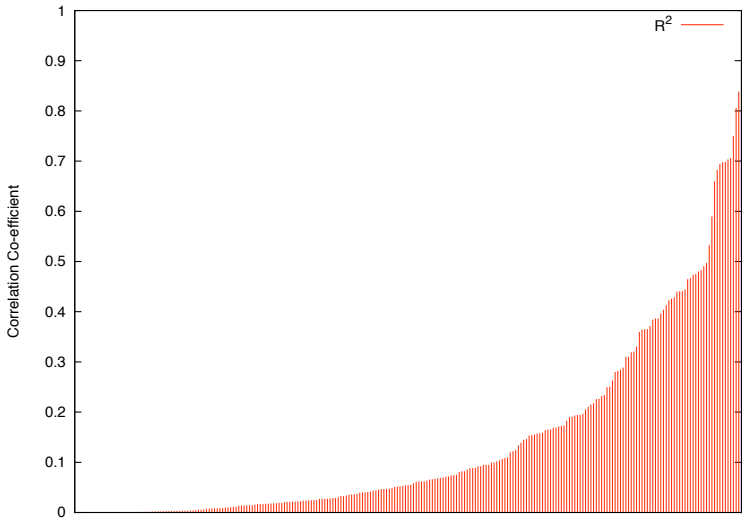

Figure 5.8: Correlation co-efficient distribution for linear regression between module MI and developers that worked on the module.

counter proofs, the most important of which, in our opinion, are the following three:

- How can researchers be confident that their theories are fit by validating them on such limited datasets?
- How can other researchers be sure that there was no bias in the experimental set selection so as to drive to research results?
- Is it possible to draw conclusions by exposing theories to very large datasets or the inherent complexity of software engineering data blurs the results and therefore all empirical research in software engineering has to be project specific?

We believe that software engineering researchers should work towards disproving their theories rather than work towards proving them. After all, the results of a study that states that a given hypothesis stands even if it has been exposed to the scrutiny of very large sets of data are more likely to become accepted than the results of a study that provides traces of the same hypothesis validity on very small datasets. SERP provides us with the opportunity to use data that more closely approximate the reality by being voluminous, diverse and presented in an analysis friendly format.

5.4 Hypotheses Validation

In this section, we examine the hypothesis we have put forward in Section 3 with respect to the results we have achieved in our work.

H1 - Software data and metadata unification We have introduced two data storage schemata, one for storing raw project data mirrors and one for storing metadata. Using the raw data schema, we were able to setup a mirroring infrastructure for more than 700 OSS projects with various sizes. The cumulative size of the data is more than 230 GB and it is constantly increasing, as the raw data are being updated daily.

More importantly, the metadata schema proved very effective in storing and retrieving metadata and linking them to raw data. As most research in software engineering is concerned with process metadata (see Section 2.3) rather than raw data, we have put particular emphasis on designing an effective intermediate storage format along with efficient algorithms to populate it. Currently, the metadata schema holds data for 530 projects, 2 million project versions, 16 million file state changes, 1.5 million emails, and 130 million measurements on all resources. It is arguably the largest collection of preprocessed software engineering data, with the possible exception of the data provided by the FlossMetrics project [GSy]. Despite being very large, it remains efficient: simple operations such as retrieving the developer that performed a commit executes instantly, while very complex operations such as retrieving the list of live files for a specific revision take time according to the project size.

Therefore, our first hypothesis can be accepted.

H2 - Effective access to metadata We have conducted two case studies that perform complex data analysis on large data sizes. In the first study, we implemented a plug-in that examines data across projects and across repositories. The rich abstractions offered by SERP coupled with the metadata schema enables the plug-in to access the processed data by means of method calls without having to perform any type of raw data parsing. The pre-computed relationships between metadata entities (for example, mailing lists threads) and the mechanism of invoking the plug-in on metadata changes, save the researcher the trouble to parse or recompute results each time the dataset is updated. Moreover, in the second case study, the implemented MI plug-in retrieves results from several other plug-ins per evaluated resource in order to calculate its value. Again this process is performed programmatically, with very minimal interaction with the metadata store.

The resulting lines of code metric for both plug-ins is indicative of the convenience that the platform provides to the researcher. To prove this claim, we compare the sizes of the code structure metrics plug-in to that of the `cmetrics` tool suite,[3] which performs exactly the same operation, namely calculates the Halstead metrics and McCabe complexity. Even though the code structure plug-in stores results in a database, can run on clusters of machines without modification and supports an extra language, the

[3] `http://tools.libresoft.es/cmetrics`

total lines of code are exactly the same. Moreover, the plug-in used by the first case study is only 270 lines of code, which is indicative of the size of a simple Alitheia Core plug-in. All sizes include comments and license headers.

Therefore, our second hypothesis can be accepted.

H3 - Efficient experimentation For both case studies we have conducted, we presented quantitative data of the performance of the developed analysis tools. The data show that the system can process large volumes of data fast, while being frugal with resources. Moreover, the system fully automates the execution of large experiments and, after the initial setup, distributes the processing load on cluster nodes. To validate the third hypothesis in full, it would be necessary to construct tools that implement exactly the same functionality as the plug-ins we implemented for each case study and run those on the same datasets. We left this part of the hypothesis validation as future work.

Therefore, our third hypothesis can be tentatively accepted.

H4 - Experiment replication So far, there was no independent replication of experiments we have performed using SERP. We believe that given the wealth of data and services SERP provides, it will be easy for other researches to replicate our experiments. However, due to the lack of available data, we are forced to leave validation of our last hypothesis an open research question.

5.5 Summary

In this section we provided an empirical validation of the usefulness of the SERP platform by conducting two exploratory case studies on real world datasets. In the first case study, we provided evidence that intense discussions do not have a significant impact on the development effort. With the second case study we correlated a process metric (developer number) with a product metric (the maintainability index) on more than two hundred projects, including some specifically large ones. We did not identify any significant correlation between the two variables, a result that might suggest that it is the developer skill or project policies and not their number that drive maintainability. We also provided evidence of why executing experiments on small data sets should be considered harmful.

Chapter 6

Conclusions and Future Work

In this dissertation, we have introduced a platform for large scale software engineering studies. We presented the components of the platform and demonstrated with two case studies that it can be used efficiently for conducting research with heterogeneous process and product data originating from diverse data sources. In this chapter, we summarize our experience and findings and describe some potential directions for future research.

6.1 Summary of Results

As with any other empirical science, research in software engineering deals with the understanding of the behaviour of existing systems, in order to extract models and best practices that can be generalised and applied to similar occasions. Currently, most research efforts in software engineering are using data from OSS projects, on a very limited scale. This fact leads to conclusions that cannot be generalised and to very limited cross-validation of the published research results.

In this context, the goals of our work were:

- to analyse the related work, in order to classify the research that has been done and identify potential weakness in the current approaches.
- to design and implement tools and methods that would allow researchers to experiment with large, heterogeneous datasets.
- to validate our proposed approach on real world scenarios.

The following sections describe how far we have reached with respect to each one of the goals set.

6.1.1 Systematic Analysis of Related Work

Empirical software engineering is a very active field of study, with the number of works being published in related conferences and journals exhibiting an increasing frequency over the last few years. This fact could be attributed to the availability of large quantities of high quality process and product data from the OSS movement.

By systematically analysing 70 works, and studying several more, we have seen that most empirical studies in software engineering are case studies that either test hypotheses on empirical data (confirmatory) or analyse the behaviour of data sets in order to extract insights, in the form of models or observations, from them (exploratory). Perhaps as a testament to the scientific method, most studies follow a certain recipe in their execution, which entails the phases of hypothesis formulation, model building and validation on real data.

What we have found from our analysis is that hypothesis validation is the weakest aspect of current studies. The problem is not that researchers do not validate their hypotheses; it lays mostly in the fact they do not do it rigorously, despite the availability of datasets. Moreover, we have also failed to see any study that validates external work. We conjecture that these shortcomings can be attributed to three important factors: the data format disparity in the OSS datasets, the fact that datasets are very large, and the lack of standardised tool chains for conducting the experiments. As a result, large experiments are expensive resource-wise to setup and time consuming to conduct.

6.1.2 Building the Platform

The bulk of our work has been devoted to the development of tools, algorithms and data formats required for storing, linking and processing very large volumes of heterogeneous OSS data. We designed a compact database schema that can integrate metadata from SCM, mailing lists and BTS databases. We designed and implemented algorithms that extract metadata from the raw data stores and populate the designed schema and also infer relationships between entities residing in the different repository types. Our intermediate schema plays a pivotal role in disengaging the design of analysis tools from the processed data formats, thereby allowing tools to be written once. This way, an important obstacle to conducting large scale studies, namely raw data format disparity, is lifted.

One of the most important contributions of our work has been the design, partially implementation and validation, of the first algorithm that converts semi-structured data from both distributed and centralised SCM repositories to a fully structured relational format. The scmmap algorithm has proven its efficiency to incorporate data from very large projects by processing the full data set from projects with hundreds of thousands of revisions featuring thousands of live files per revision.

In the process of building the platform, we have learned several lessons, the most important one is that when dealing with such large volumes of diverse data, naive and brute force approaches to data analysis do not work. No matter how simple a problem might initially seem to a developer, it is almost certain that corner cases in the processed datasets will uncover any algorithmic or data structure inefficiency. We attempt to summarise the experience we have gained from building tools and algorithms for processing over 250GB of data, while remaining scalable and extensible, in the following recommendation list:

- Understand the data. Before processing large quantities of data the researcher must fully understand their format and, especially, their interactions. This knowledge is required for designing efficient storage schemata, extracting parallelism from common operations on data or for computing the execution order of calculations. Knowledge of the data can be acquired by building prototypes or using similar tools.

- Scaling comes from architecture, not optimisation. A common misconception in engineering large systems is that performance can only be extracted by optimising algorithms or data storage. While optimisation is required for processing large data volumes, to scale to more than one machine, a system must be designed on sound architectural principles, in order to model work items, data and interactions between the machines for load distribution.

- The database does not always contain the answer. The database layer in a three tier architecture is usually the most difficult and most expensive resource to scale, and its performance profile is typically a black box for the developer. It is therefore important to limit the load that the database is exposed to. For example, complex queries can be replaced by range queries and data filtering at the requesting node cite, while appropriate indexes and data types can provide tenfold improvements in query execution speed. From our experience, a query that requires more than one second to execute should be considered as a target for optimisation. The database layer should be used only for storing and retrieving data, not for processing them.

- Fail fast, isolate failures. A high volume application should be designed to isolate and recover from errors, not to anticipate and correct them. It is usually not possible to estimate all possible system states when designing the system, and moreover not economical (in terms of development time or system performance) to develop workarounds within the system. If an unexpected error occurs, the system must be able to stop processing before errors reach persistent data stores and to resume processing from known good states.

- Manage resources internally: Large applications run on a variety of software and hardware platforms featuring different configurations and processing capacity. Our experience with SERP has shown that large scale applications exhibit more predictable behavioural patterns if they manage computing and memory resources internally. Moreover, as applications have better knowledge of the data they process, they can arrange the order of data processing without relying on expensive synchronisation mechanisms.

6.1.3 Conducting Large Scale Experiments

SERP was designed from the ground up to enable researchers to design and conduct large scale research. The platform can readily process hundreds of projects, with available hardware being currently the only obstacle towards scaling it to the order of thousands. The case studies we have presented in this thesis have evaluated an order, perhaps two, more data (both in terms of raw data sizes and in terms of project counts) than the case studies that are currently being published, and they have done so in a time constrained manner. In addition to validating our approach, this result indicates that large scale experimentation is feasible, given the appropriate hardware and data abstractions. Consequently, software engineering researchers should concentrate their efforts towards evaluating their ideas on large, real world datasets.

Apart from proving the feasibility of large scale experimentation, we have also shown the importance of such an approach. A side result of both our case studies has been the discovery of several cases where the hypotheses we rejected in the overall study could be accepted. In fact, we have found more hypothesis-validating case for each case study that the average number of projects evaluated per case study in currently published studies; we derived this number from the systematic literature review study we conducted in Chapter 2. Based on this result and acknowledging the risk of oversimplification, we conjecture that for a large number of currently published works there exists a dataset that invalidates them. We plan to work on making this finding more concrete in the future.

6.2 Future work

SERP is the first large scale, integrated software and data platform that was specifically designed for software engineering studies. While it can already process significantly large data volumes, it remains a prototype implementation and not a full blown system. In previous chapters, we provided hints on where SERP can be improved. Here, we provide a comprehensive list of future work on the platform and outline possible future research directions.

6.2.1 Data Validation

SERP integrates data from various sources and also implements a suite of metrics. How can we ensure that the data SERP produces are accurate? So far, we did not implement any validation methods in SERP. However, given our experience with various existing metric tools, we strongly believe that SERP must incorporate a validation service. The validation service can be automated for stage 1 metadata updates; we are not confident that stage 2 metadata updates (relationship inference) or metric results can be validated without human intervention. For these two cases, we advocate "crowdsourcing" [1] the validation effort. For product metrics, a web site could be set up that displays random files and corresponding metric values while for stage 2 updaters, the same web site could display the original raw data and the extracted relationship. In both cases, invited experts could examine the processing result and validate them or propose alternative values, while also enabling them to discuss with others on the same web page.

6.2.2 Results Distribution

In this work, we have discussed how SERP incorporates data into its database. Even though the system is scalable and can work with very large volumes of data, we believe that it will be very unlikely for a single installation to be able to process the full set of OSS projects, which spans to more than 300.000 projects. An alternative approach would entail the sharing of data and metadata from various smaller installations. The SQO-OSS tool, which SERP is based on, includes a programmatic interface to the metadata and results database. Unfortunately, this interface is based on the web services stack and was not designed for en-masse data retrieval. For this reason, we recommend a lightweight Representational State Transfer (REST)-based [Fie00] interface to the SERP internal database, in addition to a metadata updater that retrieves data from external SERP installations. This would create a decentralised network of peers, where data synchronisations could be performed on grounds of mutual agreements between the involved parties.

6.2.3 Repositories for Tools and Results

In this work, we have provided hints on the importance of sharing experimentation tools and data. We believe that this is a very important step towards improving the quality of software engineering studies, as it enables independent replication, avoidance of duplication of effort in re-implementing analysis tools and saves data processing time for large datasets. SERP already offers the required infrastructure to run external analysis tools, via plug-ins, and to share research results, in the form of database dumps.

[1] Distributing the processing load to interested parties over the internet

Missing are a centralised website that will store SERP plug-ins and the appropriate extentions to SERP to automate the communication with the central repository.

6.2.4 Validate Existing Work

In this thesis, we provided evidence on why doing research with small datasets, which was the major finding of our system literature review, is detrimental for the quality of the research effort. By studying the literature, we have found several suspicious cases of reported results that seem bound to the specific dataset or being extrapolated through experimentation with very small datasets. With the wealth of data present in SERP in our arsenal, we plan to perform replicated studies of published works and report our findings.

6.3 Conclusions

In this dissertation, we investigated the problem of performing empirical software engineering studies on realistic datasets from a pragmatic perspective. We analysed the requirements, the design and the several bits of the implementation of SERP, a platform that can process the SCM, mailing list and BTS data from hundreds of OSS projects, while remaining scalable and extensible. We have shown that our approach can greatly speed up the design and execution of analytical studies on large volumes of software data, while also enabling experiment replication and tool sharing.

We provide the source code we have developed as OSS software and the dataset we have produced free of charge to the community, with the ambition that both will be used by third parties to do research with. We hope that SERP will provide the basis for better empirical software engineering studies.

Bibliography

[AAB00] Bowen Alpern, C. R. Attanasio, and John J. Burton. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[ACM] ACM-SIGKDD. The knoweledge discovery and data mining cup contest.

[ADG08] Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from CVS. In *MSR '08: Proceedings of the 2008 International working conference on Mining software repositories*, pages 125–128, New York, NY, USA, 2008. ACM.

[AG83] A.J. Albrecht and Jr. Gaffney, J.E. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6):639–648, 1983.

[AHM06] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International Conference on Management of Data*, pages 207–216, 1993.

[Alb79] A. J. Albrecht. Measuring application development productivity. In *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, pages 83–92, 1979.

[AM07] John Anvik and Gail C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[ARGB06] Juan Jose Amor, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Effort estimation by characterizing developer activity. In *EDSER '06: Proceedings of the 2006 international workshop on Economics driven software engineering research*, pages 3–6, New York, NY, USA, 2006. ACM.

[ARV05] Giuliano Antoniol, Vincenzo Fabio Rollo, and Gabriele Venturi. Linear predictive coding and cepstrum coefficients for mining time variant information from software repositories. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[Bak95] Brenda S. Baker. On finding duplication and near-duplication in large software systems;. In *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, CA, USA, 1995.

[Bas96] V.R. Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th International Conference on Software Engineering*, pages 442–449, Mar 1996.

[BAY03] J.M. Bieman, A.A. Andrews, and H.J. Yang. Understanding change-proneness in OO software through visualization. pages 44–53, May 2003.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BCR94] V. Basili, C. Caldiera, and D. H. Rombach. Goal question metric paradigm. In *Encyclopedia of Software Engineering*, volume 2, pages 528–532. John Wiley and Sons, New York, 1994.

[Bev06] Jennifer Bevan. *Software Instability Analysis: Co-Change Analysis Across Configuration-Based Dependence Relationships.* PhD thesis, University of California at Santa Cruz, 2006.

[BGD+06] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143, New York, NY, USA, 2006. ACM.

[BGD+07] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? Immigration in open source projects.

In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot, B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM Press.

[BLR04] J. M. Barahona, L. Lopez, and G. Robles. Community structure of modules in the Apache project. In *Proceedings of the 4th Workshop on Open Source Software Engineering*, Edinburgh, Scotland, 2004.

[BM07] Olga Baysal and Andrew J. Malton. Correlating social interactions to release history during software evolution. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[BMB96] LC Briand, S. Morasca, and VR Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, 1996.

[BN05] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proceedings. 13th International Workshop on Program Comprehension*, pages 259–268, May 2005.

[BRB+09] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining Git. In M. W. Godfrey and J Whitehead, editors, *MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on Mining Software Repositories*, pages 1–10, Vancouver, Canada, 2009.

[BRM04] J. Bowring, J. Rehg, and Harrold M.J. Active learning for automatic classification of software behavior. *International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[Bro82] Fred. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison Wesley, Reading, Mas, 1982.

[BSH86] Victor Basili, RW Selby, and DH Hutchens. Experimentation in software engineering. *IEEE Trans. Softw. Eng.*, 12(7):733–743, 1986.

[BvDTvE04] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen. An evaluation of clone detection techniques for crosscutting concerns. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 200–209, Sept. 2004.

[BWKG05] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with Kenyon. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 177–186, New York, NY, USA, 2005. ACM.

[BYM+98] ID Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, S. Designs, and TX Austin. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[CALO94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[CC05] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. pages 9 pp.–29, Sept. 2005.

[CC06] Gerardo Canfora and Luigi Cerulo. Fine grained indexing of software repositories to support impact analysis. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 105–111, New York, NY, USA, 2006. ACM.

[CCW+01] A. Chen, E. Chou, J. Wong, A.Y. Yao, Qing Zhang, Shao Zhang, and A. Michail. CVSSearch: searching through source code using CVS comments. In *Proceedings of the IEEE International Conference on Software Maintenance, 2001.*, pages 364–373, 2001.

[CG90] David N. Card and Robert L. Glass. *Measuring software design quality.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[CH05] K. Crowston and J. Howison. The social structure of free and open source software development. *Firstmonday*, 10(2), 2005.

[Cha95] Vernon V. Chatman. Change-points: A proposal for software productivity measurement. *Journal of Systems and Software*, 31(1):71 – 91, 1995.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), Jun 1994.

[CM03] D. Cubranic and G.C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, May 2003.

[CMR04a] A. Capiluppi, M. Morisio, and J.F. Ramil. The evolution of source folder structure in actively evolved open source systems. In *Proceedings of the 10th International Symposium on Software Metrics, Chicago, USA*, pages 2–13, 2004.

[CMR04b] A. Capiluppi, M. Morisio, and J.F. Ramil. Structural evolution of an open source system: a case study. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 172–182, June 2004.

[CMSB05] D. Cubranic, G.C. Murphy, J. Singer, and K.S. Booth. Hipikat: a project memory for software development. *Software Engineering, IEEE Transactions on*, 31(6):446–465, June 2005.

[Coa04] Ken Coar. The sun never sits on distributed development. *Queue*, 1(9):32–39, 2004.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cro82] David H. Crocker. Standard for the format of arpa internet text messages. RFC 822, Internet Engineering Task Force, 1982.

[CSX+06] H. Chen, H. Shen, J. Xiong, S. Tan, and X. Cheng. Social network structure behind the mailing lists: Ict-iiis at trec 2006 expert finding track. In *Proceeddings of the 15th Text Retrieval Conference*, 2006.

[Der94] Mark Dery, editor. *Flame Wars: The Discourse of Cyberculture*. Duke University Press, 1994.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[DJ03] Melis Dagpinar and Jens H. Jahnke. Predicting maintainability with object-oriented metrics - an empirical comparison. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 155, Washington, DC, USA, 2003. IEEE Computer Society.

[DLA01] W. Dickinson, D. Leon, and Podgurski A. Finding failures by cluster analysis of execution profiles. *International Conference on Software Engineering (ICSE)*, 2001.

[DR08] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 481–490, New York, NY, USA, 2008. ACM.

[DRD99] Stephane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance*, pages 109–118, Sep 1999.

[ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. *Guide to Advanced Empirical Software Engineering*, chapter Selecting Empirical Methods for Software Engineering Research, pages 285–311. Springer London, 2008.

[FB96] N. Freed and N. Borenstein. RFC2045 - multipurpose internet mail extensions (MIME). Technical report, Internet Engineering Task Force, Nov 1996.

[FCS+08] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 261–270, New York, NY, USA, 2008. ACM.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California at Irvine, 2000.

[FL78] Ann Fitzsimmons and Tom Love. A review and evaluation of software science. *ACM Comput. Surv.*, 10(1):3–18, 1978.

[FLMP04] P. Francis, D. Leon, M. Minch, and A. Podguraki. Tree-based method for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.

[Fog99] Karl Franz Fogel. *Open Source Development with CVS*. Coriolis Group Books, Scottsdale, AZ, USA, 1999.

[Fog05] Karl Fogel. *Producing Open Source Software*, pages 261–268. O'Reilly Media, Inc, Sebastopol, 2005.

[FORG05] Michael Fischer, Johann Oberleitner, Jacek Ratzinger, and Harald Gall. Mining evolution data of a product family. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FP98] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology, 1998.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[FPSS+96] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, et al. From data mining to knowledge discovery in databases. *Communications of the ACM*, 39(11):24–26, 1996.

[FRLWC08] Juan Fernandez-Ramil, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi. *Software Evolution*, chapter Empirical Studies of Open Source Evolution. Number 11. Springer Berlin Heidelberg, 2008.

[FSG04] R. Ferenc, I. Siket, and T. Gyimothy. Extracting facts from open source software. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 60–69, Sept. 2004.

[FSGK06] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Effective identification of source code authors using

byte-level information. In *Proceedings of the 28th International conference on Software engineering*, pages 893–896, New York, NY, USA, 2006. ACM.

[Ger04a] Daniel M. German. An empirical study of fine-grained software modifications. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 316–325, Washington, DC, USA, 2004. IEEE Computer Society.

[Ger04b] Daniel M. German. Mining CVS repositories, the Softchange experience. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 17–21, Edinburg, Scotland, UK, 2004.

[Ger05] Daniel M. German. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2005.

[Ger06] D. M German. A study of the contributors of PostgreSQL. In *MSR '06: Proceedings of the 2006 international workshop on Mining Software Repositories*, pages 163–164, May 2006.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, Oct. 2005.

[GH05] Daniel M. German and Abram Hindle. Measuring fine-grained change in software: Towards modification-aware change metrics. *IEEE International Symposium on Software Metrics*, 0:28, 2005.

[GHJ98] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the 14th IEEE International Conference in Software Maintainance*, pages 190–198, 1998.

[GJK03] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution*, pages 13–23, Sept. 2003.

[GJR99] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: the use of color and thirddimension. In *Proceedings of the IEEE International Conference on Software Maintenance, 1999*, pages 99–108, 1999.

[GKS+07] Georgios Gousios, Vassilios Karakoidas, Konstantinos Stroggylos, Panagiotis Louridas, Vasileios Vlachos, and Diomidis Spinellis. Software quality assessment of open source software. In *Proceedings of the 11th Panhellenic Conference on Informatics*, May 2007.

[GM03] D. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 63–67, 2003.

[GM07] Yongqin Gao and Greg Madey. *Network Analysis of the SourceForge.net Community*. Springer Boston, 2007.

[GPGA09] Daniel M. German, Massimiliano Di Penta, Yann-Gaël Gueheneuc, and Giuliano Antoniol. Code siblings: Technical and legal implications of copying code between applications. In Michael W. Godfrey and Jim Whitehead, editors, *MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on Mining Software Repositories*, Vancouver, Canada, 2009. IEEE.

[GRW08] Thomas Goldschmidt, Ralf Reussner, and Jochen Winzen. A case study evaluation of maintainability and performance of persistency techniques. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 401–410, New York, NY, USA, 2008. ACM.

[GS09] Georgios Gousios and Diomidis Spinellis. Alitheia core: An extensible software quality monitoring platform. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering – Formal Research Demonstrations Track*, pages 579–582. IEEE, May 2009.

[GSy] GSyC/LibreSoft. The FlossMetrics project.

[GT00] MW Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142, 2000.

[GVG04] J.F. Girard, M. Verlage, and D. Ganesan. Monitoring the evolution of an OO system with metrics: an experience from the stock market software domain. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 360–367, Sept. 2004.

[GVR02] R. L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44(8):491 – 506, 2002.

[GW05] C. Gorg and P. Weissgerber. Detecting and visualizing refactorings from software archives. pages 205–214, May 2005.

[Hal77] M.H. Halstead. *Elements of software science*. Elsevier Publishing Company, 1977.

[Hat98] L Hatton. Does OO sync with how we think? *IEEE Software*, 15:3, 1998.

[HCC06] J. Howison, M. Conklin, and K. Crowston. Flossmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[Her08] Israel Herraiz. *A statistical examination of the properties and evolution of libre software*. PhD thesis, Universidad Rey Juan Carlos, 2008.

[HH04] A.E. Hassan and R.C. Holt. Predicting change propagation in software systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Sept. 2004.

[HK81] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518, 1981.

[HM03] J.D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *Software Engineering, IEEE Transactions on*, 29(6):481–494, June 2003.

[HRA+06] Israel Herraiz, Gregorio Robles, Juan JosÉ Amor, Teófilo Romera, and Jesús M. González Barahona. The processes of joining in global distributed software projects. In *GSD '06: Proceedings of the 2006 international workshop on Global software development for the practitioner*, pages 27–33, New York, NY, USA, 2006. ACM.

[HT02] A. Hunt and D. Thomas. Software archaeology. *Software, IEEE*, 19(2):20–22, Mar/Apr 2002.

[IEE90] IEEE. IEEE standard glossary of software engineering terminology. IEEE std 610.12, IEEE, Dec 1990.

[ISO04] ISO/IEC. 9126:2004 Software engineering – Product quality – Quality model. Technical report, International Organization for Standardization, Geneva, Switzerland, 2004.

[JKP+05] P.M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *Software, IEEE*, 22(4):76–85, July-Aug. 2005.

[Joh93] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the IBM Centre from Advanced Studies Conference*, pages 171–183, 1993.

[Jon91] T. C. Jones. *Applied Software Measurement*. McGraw-Hill, New York, 1991.

[Kan03] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley Professional, 2003.

[KCM07a] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. Comparing approaches to mining source code for call-usage patterns. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 20, Washington, DC, USA, 2007. IEEE Computer Society.

[KCM07b] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution*, 19(2):77–131, Mar 2007.

[KDTM06] Y. Kanellopoulos, T. Dimopulos, C. Tjortjis, and C. Makris. Mining source code elements for comprehending object-oriented systems and evaluating their maintainability. *ACM SIGKDD Explorations Newsletter*, 8(1):33–40, 2006.

[Kee94] S. J. Keene. Comparing hardware and software reliability. *Reliability Review*, 14(4):5–7, Dec 1994.

[KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis,*, pages 40–56, 2001.

[KHM08] H. Kagdi, M. Hammad, and J.I. Maletic. Who can help me with this source code change? In *IEEE International Conference on Software Maintenance*, pages 157–166, 28 2008-Oct. 4 2008.

[Kit04] B Kitchenham. Procedures for performing systematic reviews. Technical report, Software Engineering Group, Keele University, United

Kingdom and Empirical Software Engineering, National ICT Australia Ltd, Australia, 2004.

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[KN05] Miryung Kim and David Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[KPP+02] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734, Aug 2002.

[KPW06] Sunghun Kim, Kai Pan, and E. James Whitehead, Jr. Micro pattern evolution. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 40–46, New York, NY, USA, 2006. ACM.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eight Working Conference On Reverse Engineering*, pages 101–109, Oct 2001.

[KWB05] Sunghun Kim, E. James Whitehead, and Jennifer Bevan. Analysis of signature change patterns. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[kwW97] Marvin V. Zel ko witz and Dolores Wallace. Experimental validation in software engineering. *Information and Software Technology*, 39(11):735 – 743, 1997. Evaluation and Assessment in Software Engineering.

[KYM06] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining sequences of changed-files from version histories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 47–53, New York, NY, USA, 2006. ACM.

[KZPW06] S. Kim, T. Zimmermann, K. Pan, and EJ Whitehead. Automatic

identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE'06*, pages 81–90, 2006.

[Lai06] Linda Laird. *Softare Measurement and Estimation: A Practical Approach.* John Wiley & Sons, Inc, 2006.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[LFK05] M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *Proceeding of the SIGKDD Conference*, 2005.

[LM03] Giovani F. Lanzara and Michäle Morner. The knowledge ecology of open-source software projects. In *Proceedings the 19th European Group of Organizational Studies Colloquium.* European Group of Organizational Studies, 2003.

[LRB06] F. Lopez, G. Robles, and J. Barahona. Applying social network analysis techniques to community-driven libre software projects. *International Journal of Information Technology and Web Engineering*, 1(3):27–48, 2006.

[LRW+97] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski. Metrics and laws of software evolution-the nineties view. pages 20–32, Nov 1997.

[LSS05] Timothy C. Lethbridge, Susan Elliott Sim, and Janice Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341, 2005.

[LYY+05] C. Liu, X. Yan, H. Yu, J. Han, and P. Yu. Mining behavior graphs from 'backtrace' of non-crasing bugs. In *SIAM Data Mining Conference (SDM)*, 2005.

[LZ05] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[Mas05] Bart Massey. Longitudinal analysis of long-timescale open source repository data. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[McC76] TJ McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, pages 308–320, 1976.

[MFH02] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[MFT02] G. Madey, V. Freeh, and R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Americas Conference on Information Systems*, pages 1806–1813, 2002.

[MINK07] Osamu Mizuno, Shiro Ikami, Shuya Nakaichi, and Tohru Kikuno. Spam filter based approach for finding fault-prone software modules. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 4, Washington, DC, USA, 2007. IEEE Computer Society.

[Mis05] Subhas Chandra Misra. Modeling design/coding factors that drive maintainability of software systems. *Software Quality Control*, 13(3):297–320, 2005.

[MLM96] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, 1996.

[MM07] Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 5, Washington, DC, USA, 2007. IEEE Computer Society.

[MMM+07] Shuji Morisaki, Akito Monden, Tomoko Matsumura, Haruaki Tamada, and Ken-ichi Matsumoto. Defect data analysis based on extended association rule mining. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[Moc07] A. Mockus. Large-scale code reuse in open source software. *First International Workshop on Emerging Trends in FLOSS Research and Development*, pages 7–7, May 2007.

[Moc09] Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In

M. W. Godfrey and J. Whitehead, editors, *MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on Mining Software Repositories*, pages 11–20, 2009.

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 181–190, New York, NY, USA, 2008. ACM.

[MRRGBOP08] Juan Martinez-Romo, Gregorio Robles, Jesus M. Gonzalez-Barahona, and Miguel Ortuño-Perez. *Open Source Development, Communities and Quality*, volume 275 of *IFIP International Federation for Information Processing*, chapter Using Social Network Analysis Techniques to Study Collaboration between a FLOSS Community and a Company, pages 143–158. Springer Boston, Jul 2008.

[MSCC04] Tim Menzies, Justin S. Di Stefano, Chris Cunanan, and Robert (Mike) Chapman. Mining repositories to assist in project planning and resource allocation. In *Proceedings of the 1rst Workshop on Mining Software Repositories*, 2004.

[Mut04] Paul Mutton. Inferring and visualizing social networks on internet relay chat. *Information Visualisation, International Conference on*, 0:35–43, 2004.

[MV00] A. Mockus and L.G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings. International Conference on Software Maintenance*, pages 120–130, 2000.

[MVL03] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance*, pages 381–384, Sept. 2003.

[MVL04] M.V. Mantyla, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 399–408, Sept. 2004.

[MWZ03] A. Mockus, D.M. Weiss, and Ping Zhang. Understanding and predicting effort in software projects. *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 274–284, May 2003.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 452–461, New York, NY, USA, 2006. ACM.

[NM03] A.P. Nikora and J.C. Munson. Understanding the nature of software evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 83–93, Sept. 2003.

[OH94] P. Oman and J. Hagemeister. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(251–266), 1994.

[O'N08] Elizabeth J. O'Neil. Object/relational mapping 2008: Hibernate and the entity data model (edm). In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356, New York, NY, USA, 2008. ACM.

[OOOM05] Masao Ohira, Naoki Ohsugi, Tetsuya Ohoka, and Ken-ichi Matsumoto. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[OSG07] *OSGi Service Platform, Core Specification.* OSGi Alliance, 2007.

[OW04] Thomas J. Ostrand and Elaine J. Weyuker. A tool for mining defect-tracking systems to predict fault-prone files. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Edinburg, Scotland, UK, 2004.

[Par58] C. Northcote Parkinson. *Parkinson's Law: The Pursuit of Progress.* John Murray, 1958.

[Par92] Robert E. Park. Software size measurement: a framework for counting source statements. Technical Report CMU/SEI-92-TR-020, Software Engineering Institute, 1992.

[Par94] D.L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, May 1994.

[PCSF08] C Pilato, Ben Collins-Sussman, and Brian Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., 2nd edition, 2008.

[Pel01] Doron A. Peled. *Software Reliability Methods*. Springer-Verlag, 2001.

[PG08] Chris Parnin and Carsten Görg. Improving change descriptions with change contexts. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 51–60, New York, NY, USA, 2008. ACM.

[Phi00] Lawrence Phillips. The double metaphone search algorithm. *C/C++ Users Journal*, 2000.

[PMM+03] A. Podgurski, W. Masri, Y. McCleese, M. Minch, J. Sun, B. Wang, and W. Masri. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[Pop35] Karl Popper. *The Logic Of Scientific Discovery*. 1935.

[PP05] R. Purushothaman and D.E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, June 2005.

[PPV00] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 345–355, New York, NY, USA, 2000. ACM.

[PSE04] J.W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *Software Engineering, IEEE Transactions on*, 30(4):246–256, April 2004.

[RGB05] Gregorio Robles and Jesus M. Gonzalez-Barahona. Developer identification methods for integrated data from various sources. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[RGBH05] G. Robles, J.M. Gonzalez-Barahona, and I. Herraiz. An empirical approach to Software Archaeology. In *Proceedings of the International Conference on Software Maintenance (Poster Session)*, 2005.

[RGBM06] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Juan Julian Merelo. Beyond source code: The importance of other artifacts in

software development (a case study). *Journal of Systems and Software*, 79(9):1233 – 1248, 2006. Selected papers from the fourth Source Code Analysis and Manipulation (SCAM 2004) Workshop.

[RGBMA06] Gregorio Robles, Jesus M. Gonzalez-Barahona, Martin Michlmayr, and Juan Jose Amor. Mining large software compilations over time: another perspective of software evolution. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 3–9, New York, NY, USA, 2006. ACM.

[RH83] Andreas Reuter and Theo Haerder. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, Dec 1983.

[RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, March 2009.

[RKGB04] Gregorio Robles, Stefan Koch, and Jesus M. Gonzalez-Barahona. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Edinburg, Scotland, UK, 2004.

[Rob05] Gregorio Robles. *Empirical Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, Madrid, 2005.

[Roc75] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[RRL+04] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 188–197, Sept. 2004.

[RRM04] A.J. Rostkowycz, V. Rajlich, and A. Marcus. A case study on the long-term effects of software redocumentation. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 92–101, Sept. 2004.

[RSG99] Linda H. Rosenberg, Ruth Stapko, and Al Gallo. Applying object-oriented metrics. In *Proceedings of the Sixth International Symposium*

on Software Metrics - Workshop on Measurement for Object-Oriented Software, Boca, Raton, FL, 1999.

[RSG08] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 35–38, New York, NY, USA, 2008. ACM.

[Sca04] W. Scacchi. Free and open source development practices in the game community. *Software, IEEE*, 21(1):59–66, Jan-Feb 2004.

[SDJ07] Dag I.K Sjøberg, Tore Dyba, and Magne Jorgensen. The future of empirical methods in software engineering research. In *FOSE '07: 2007 Future of Software Engineering*, pages 358–378, Washington, DC, USA, 2007. IEEE Computer Society.

[Sea99] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *Software Engineering, IEEE Transactions on*, 25(4):557–572, Jul/Aug 1999.

[SEG68] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM*, 11(1):3–11, 1968.

[SEI04] Maintainability index technique for measuring program maintainability. Online, 2004.

[Sha03] M. Shaw. Writing good software engineering research papers. In *Proceedings of the 25th International Conference on Software Engineering, 2003.*, pages 726–736, May 2003.

[SHH+05] Dag I.K Sjøberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A.C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, Sept. 2005.

[SI94] M. Shepperd and D.C. Ince. A critique of three metrics. *Journal of Systems Software*, 26:197–210, 1994.

[SJAH09] Weiyi Shang, Zhen Ming Jiang, Bram Adams, and Ahmed E. Hassan. Mapreduce as a general framework to support research in mining software repositories. In Michael W. Godfrey and Jim Whitehead, editors, *MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on*

Mining Software Repositories, pages 21–30, Vancouver, Canada, 2009 2009. IEEE.

[SJH09] Emad Shihab, Zhen Ming Jiang, and Ahmed E. Hassan. On the use of internet relay chat meetings by developers of the GNOME GTK+ project. In M. W. Godfrey and J. Whitehead, editors, *MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on Mining Software Repositories*, pages 107–111, Vancouver, Canada, May 2009. IEEE.

[SJM08] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 471–480, New York, NY, USA, 2008. ACM.

[SJW+02] SR Schach, B. Jin, DR Wright, GZ Heller, and AJ Offutt. Maintainability of the Linux kernel. *IEE Proceedings-Software*, 149(1):18–23, 2002.

[SL01] Jelber Sayyad and C. Lethbridge. Supporting software maintenance by mining software update records. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 22, Washington, DC, USA, 2001. IEEE Computer Society.

[Sow07] Sulayman K. Sowe. *An Empirical Study of Knowledge Sharing in Free and Open Source Software Projects*. PhD thesis, Aristotle University of Thessaloniki, 2007.

[Spi06a] Diomidis Spinelis. Global software development in the FreeBSD project. In P. Kruchten, Y. Hsieh, E. MacGregor, D. Moitra, and W. Strigel, editors, *International Workshop on Global Software Development for the Practitioner*, pages 73–79. ACM Press, May 2006.

[Spi06b] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley, Boston, MA, 2006.

[Spi08] Diomidis Spinellis. A tale of four kernels. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 381–390, New York, NY, USA, 2008. ACM.

[SRB+08] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *ICSE '08: Proceedings*

of the 30th international conference on Software engineering, pages 251–260, New York, NY, USA, 2008. ACM.

[SSA06] Sulayman Sowe, Ioannis Stamelos, and Lefteris Angelis. Identifying knowledge brokers that yield software engineering knowledge in oss projects. *Information and Software Technology*, 48(11):1025–1033, November 2006.

[SSAO04] Ioannis Samoladas, Ioannis Stamelos, Lefteris Angelis, and Apostolos Oikonomou. Open source software development should strive for even greater code maintainability. *Commun. ACM*, 47(10):83–87, 2004.

[SZ08] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124, New York, NY, USA, 2008. ACM.

[SZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[TLPH95] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9 – 18, 1995.

[TMK+06] M. Tsunoda, A. Monden, T. Kakimoto, Y. Kamei, and K Matsumoto. Analyzing OSS developers' working time using mailing lists archives. In *MSR '06: Proceedings of the 2006 International workshop on Mining software repositories*, pages 181–182. ACM, 2006.

[Ven06] Gina Venolia. Textual alusions to artifacts in software-related repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 151–154, New York, NY, USA, 2006. ACM.

[VRD04] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. pages 328–337, Sept. 2004.

[VTG+06] S. Valverde, G. Theraulaz, J. Gautrais, V. Fourcassie, and R.V. Sole. Self-organization patterns in wasp and open source communities. *Intelligent Systems, IEEE*, 21(2):36–40, March-April 2006.

[WCN08] Gursimran Singh Walia, Jeffrey C. Carver, and Nachiappan Nagappan. The effect of the number of inspectors on the defect estimates produced by capture-recapture models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 331–340, New York, NY, USA, 2008. ACM.

[Wey88] E.J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[WF94] Stanley Wasserman and Kathrine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[WH05a] C.C. Williams and J.K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, June 2005.

[WH05b] Chadd C. Williams and Jeffrey K. Hollingsworth. Recovering system specific rules from software repositories. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[WM07] Michael Weiss and Gabriella Moroiu. *Emerging Free and Open Source Software Practices*, chapter Ecology and Dynamics of Open Source Communities. IGI Global, 2007.

[WND08] Peter Weissgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76, New York, NY, USA, 2008. ACM.

[WO95] K.D. Welker and P.W Oman. Software maintainability metrics models in practice. *Journal of Defence Software Engineering*, 8(19–23), 1995.

[WPZZ07] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, Washington, DC, USA, 2007. IEEE Computer Society.

[WW00] C. Wohlin and A. Wesslen. *Experimentation in software engineering – An introduction*. Kluwer Academic Publishers, 2000.

[WZX+08] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language

and execution information. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 461–470, New York, NY, USA, 2008. ACM.

[XGCM05] Jin Xu, Yongqin Gao, S. Christley, and G. Madey. A topological analysis of the open souce software development community. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, Jan. 2005.

[YMNCC04] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, Sept. 2004.

[ZMM06] Carmen Zannier, Grigori Melnik, and Frank Maurer. On the success of empirical studies in the international conference on software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2006. ACM.

[ZW04] T. Zimmerman and P Weissgerber. Preprocessing cvs data for fine-grained analysis. In *Proceeding of the 1rst Workshop on Mining Software Repositories*, pages 2–6, 2004.

[ZZWD05] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

Acronyms

ACM Association for Computing Machinery

AST Abstract Syntax Tree

API Application Programming Interface

BMI Backlog Management Index

BTS Bug Tracking Systems

CBO Coupling Between Objects

CVS Concurrent Version System

CK Chidamber and Kemerer

DBMS Database Management System

DIT Depth of Inheritance Tree

DRE Defect Removal Effectiveness

EMCC Extended McCabe Cyclomatic Complexity

ECT Electronic Communication Trail

FSF Free Software Foundation

GQM Goal Question Metric

HV Halstead Volume

HTML HyperText Markup Language

IEEE Institute of Electical and Elecronic Engineering

IF Information Flow

IM Instant Messaging

IRC Instant Relay Chat

KDD Knoweledge Discovery in Databases

LCOM Lack of Cohesion in Methods

LOC Lines of Code

MI Maintainability Index

MIME Multipurpose Internet Mail Extensions

MCC McCabe Cyclomatic Complexity

MSR Mining Software Repositories

MTTR Mean Time To Repear

MARC Mail Archive

NOC Number of Children

ORM Object Relational Mapping

OSS Open Source Software

REST Representational State Transfer

RFC Response For Class

RHDB Release History DataBase

SCM Software Configuration Management

SCCS Source Code Control System

SERP Software Engineering Research Platform

SF SourceForge

SNA Social Network Analysis

SQO-OSS Software Quality Observatory for Open Source Software

SVN Subversion

URL Universal Resource Locator

WMC Weighted Methods per Class

XML eXtensible Markcup Language