NAIST-IS-DD1061028

Doctoral Dissertation

Quantitative Analysis of Maintenance Processes at the Micro Level

Raula Gaikovina Kula

March 15, 2013

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Raula Gaikovina Kula

Thesis Committee:

Professor Hajimu Iida	(Supervisor)
Professor Ken-ichi Matsumoto	(Co-supervisor)
Professor Hiroyuki Seki	(Co-supervisor)
Assistant Professor Norihiro Yoshida	(Co-supervisor)

Quantitative Analysis of Maintenance Processes at the Micro Level*

Raula Gaikovina Kula

Abstract

The current state of mining software repository tools and technologies has provided opportunities for quantitative studies in software engineering. In this dissertation, these mined data are used to reconstruct the micro processes performed daily by developers (referred to as Micro Process Analysis (MPA)). We investigated how MPA complements the current software process improvement (SPI) initiatives. Unlike typical macro level SPI models, we demonstrated the application of MPA at the maintenance phase. Specifically, we targeted micro processes associated with bug & patch resolution and peer review.

For bug and patch processes, we quantitatively re-established Lehman's law between maintenance effort and code complexity. With three open source software (OSS) projects and a closed experiment, our proposed metrics proved this relationship to be statistically significant.

For peer review processes, we developed two models to assist OSS members identifying their social standing and career trajectory. SPI is achieved by more efficient and higher quality reviews, through the identification of expertise. Providing a career trajectory model encourages member participation, thus it promotes the sustainability of peer reviews within a project.

Our techniques and approaches validated the application of MPA for software maintenance. We concluded that micro processes could serve as supplements to macro processes, therefore providing an *'added dimension'* to SPI.

Keywords:

Software Process Improvement, Mining Software Repositories, Software Metrics

* Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD1061028, March 15, 2013.

マイクロレベルでのメンテナンスプロセスの定量的分析*

ラウラ・ガイコビナ・クーラ

内容梗概

ソフトウェアリポジトリマイニングやツールの登場は，ソフトウェア工学分野において定量的な研究の機会を提供している．本研究では，マイクロプロセス分析と呼ばれる手法を用いて，マイニングされたデータから開発者によって日々行われているマイクロプロセスを再構築した．これを用いてマイクロプロセス分析が現在のSPI (Software Process Improvement) をどのように補完しているかについて調査を行った．本論文では、典型的なマクロレベルでのSPI モデルとは異なる手法としてのマイクロプロセス分析をメンテナンスプロセスに対して適用した．特にバグ修正，パッチ適用およびピアレビューと関連したマイクロプロセスを対象とした．バグ修正，パッチ適用プロセスにおいては，保守に必要な作業量とコードの複雑性との間でLehmanの法則を定量的に再検証した。また、３つのオープンソースソフトウェア（OSS）プロジェクトでの検証および実験を行い，提案したメトリクスが前述の関係を統計的に満たすことを示した．ピアレビュープロセスにおいては，OSS開発者のコミュニティー内での地位や経歴の特定を支援することを目的として２つのモデルを構築した．本モデルに基づいて開発者の専門知識を特定することで，より効率的で高品質なレビューを実現することが可能となる。また、経歴モデルを提供し，開発者のプロジェクトへの参加を促すことで，プロジェクトの持続可能性を向上させることも期待できる．以上のように、本研究で提案した技術と手法によってソフトウェアの保守工程におけるマイクロプロセス分析の適用を検証した結果、マイクロプロセスはマクロプロセスの補助としての役割を果たし，SPIに対して新たな側面を提供することが可能であることを確認した

* 奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DD1061028, 2013年03月15日.

キーワード

Software Process Improvement, リポジトリマイニング, ソフトウェアメトリクス

List of Major Publications

1. Raula Gaikovina Kula, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida, *"Micro Process Analysis of Maintenance Effort: An OSS Case Study using Metrics based on Program Slicing,"* Journal of Software: Evolution and Process. (to appear)

2. Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, Xin Yang, Hajimu Iida: *"Using Profiling Metrics to Categorise Peer Review Types in the Android Project,"* In Supplemental Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), pp.146-151, Dallas, TX, USA, November 2012.

3. Raula Gaikovina Kula, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida, *"Experimental Study of Quantitative Analysis of Maintenance Effort using Program Slicing-based Metrics,"* In Proceedings of International Workshop on Software Quality and Management (SQAM 2012), pp.50-57, Hong Kong, 2012.

4. Raula Gaikovina Kula, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida, *"Using Program Slicing Metrics for the Analysis of Code Change Processes,"* In Proceedings of 2010 International Workshop on Empirical Software Engineering in Practice (IWESEP 2010), pages 53-58, December 2010.

5. Raula Gaikovina Kula, Kyohei Fushida, Shinji Kawaguchi, and Hajimu Iida, *"Analysis of Bug Fixing Processes Using Program Slicing Metrics,"* In

Proceedings of the 11th Product-Focused Software Process Improvement (PROFES 2010), LNCS 6156, pages 032-046, June 2010.

Other Related Publications

1. Raula Gaikovina Kula, Hajimu Iida, "Bug Fixing Process Analysis Using Program Slicing Techniques," 6th Forum on Reliable Computer SoftwarE (FORCE10), 2010, pages 73-80, Mar 2010.

2. Xin Yang, Raula Gaikovina Kula, Camargo Cruz Ana Erika, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, and Hajimu Iida, *"Understanding OSS Peer Review Roles in Peer Review Social Network (PeRSoN),"* In Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC2012) pp.709-712, Hong Kong, 2012.

3. Kazuki Hamasaki, Kenji Fujiwara, Norihiro Yoshida, Raula Gaikovina Kula, Kyohei Fushida and Hajimu Iida, *"Analysis of Patch Reviews in the Android Open Source Project,"* The Special Interest Group Technical Reports of IPSJ, Vol.2012-SE-176, No. 12, 2012.

Acknowledgements

During my PhD. journey, had I not being blessed with much guidance, inspiration and friendship, I would not have been able to overcome all obstacles.

First and foremost, to Professor Hajimu Iida, who is the backbone of my inspiration to study for my doctorate and my stay in Japan. As a fatherly figure, since 2007 he guided and advised me both in and outside of the lab, making me feel most at home. Thank you. To my thesis committee. Professor Ken-ichi Matsumoto, you have seen my work evolve from my very first master's seminar till my final defence. Thank you for your advice. Additionally, thank you Professor Hiroyuki Seki for your valuable comments and clarifications. You give me more confidence and added the quality touch to my work.

Yoshida-sensei my advisor and mentor, I enjoyed our discussions in japan-lish and look forward to more work to be done. Including Fushida-san, you both instilled much needed confidence and enstowed belief in myself. Thank you so much. Erika-sensei, you inspired me to do more.

To all my lab members of Software Design, both current and past. Each person has always inspired me with their work ethic and friendliness. Comments both large and small and a friendly environment always helps inspire the mind. Thank you so much.

My international community in NAIST, Hamada-san, Ohta-san and Yurie, thank you so much over the years. To all current and past students, you all contributed in one way or another. I am grateful to the MEXT scholarship for giving me the opportunity to study and the people I have met in Japan. Special thanks to my PNG colleagues in Japan. Martha, David, Solo, thank you. Ori, you give me much needed PNG comradeship. Thank you.

I proud to say that I am adopted Filipino. The first people (Bert & Za, Eds

& Jo) overloaded me with all advice and kept me close. Thanks for the friday talks and sharing your lives. Over the years even if the people have changed, the homely feeling of family has never eluded. Arashi girls, geek guys and everyone in between I thank you all. Jimlyns & kuro thanks for the company.

To my grandis guys, the pages do not do justice for your role over the years. Its like relaxation and inspiration with a little bit of craziness. I believe we needed that to ease the stress and hassles of research. Akie, lighten up bro. JC, fun times. Emarc & Clare clare clare, big hugs all around. Through the hardships, frustrations, joys and just crazy moments. I carry a part of you guys with me. Thank you guys.

I will never forget the our countless moments. Rusty, my partner in crime, you kept me going, even if my batteries were down. CC, my cheerleader and light during the dark moments. Thanks for standing beside me. May more adventures follow.

Finally, I give thanks to my bloodlines that stretch from Hula all the way to the tiny point of Paramana, Papua New Guinea. Win and family, Mota and family, Kukz, Gaiko and G-Rox, you fueled my engine with constant belief.

Finally, I dedicate this thesis for Mummy and Daddy, my very own super heroes. The dream all started with you and this is your accomplishment. I thank our lord for being blessed with such loving and inspiring senseis from the very beginning.

Tanique maparamu.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advancements in data repository, mining tools and techniques have resulted in substantial automatic archiving of the fine-grained activities of developers. For instance, source code repository management technologies such as GIT [1] and SubVersion (SVN)[2] rival the shortcomings of the traditional Concurrent Versioning System (CVS) [3] . Also, improvements in linkages between both issue tracking and peer review management systems with source code repositories have enabled more precise and detailed reconstruction of daily activities of developers during software development. This has led to an emergence of empirical analysis, especially in the sub-fields of Software Engineering such as *mining software repositories* (MSR) and *software maintenance* (SM).

Practitioners and researchers alike understand the importance of Software Process Improvement (SPI) for software development. State-of-the-art SPI models are mostly at project level and are driven by business goals. As seen in Fig. 1.1, most SPI initiative are at the macro level (top half) and are driven by external factors such as the number of developers, the project schedule and costs.

Studies have highlighted incompatibility issues of SPI initiatives with smaller organisations, especially in terms of complexity and implementation costs [105, 43, 12, 93, 83, 30]. Other perils such as the accuracy and integrity of the manually recorded data is questionable. Additionally, most models purposely cover all

[1] http://git-scm.com/

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

[3] http://www.nongnu.org/cvs/

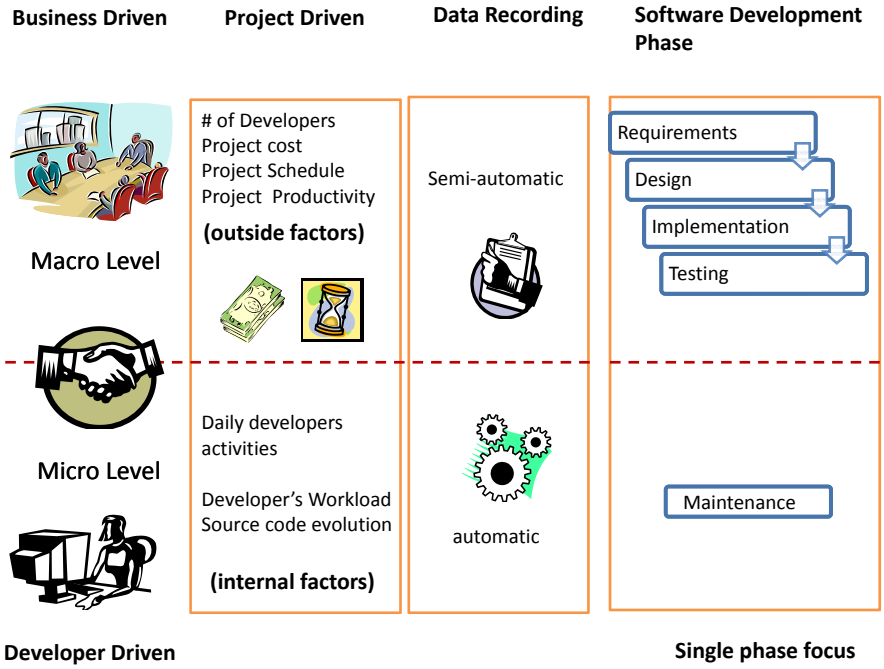


Figure 1.1: Quantitative data collected during software development at the Macro (top half) and Micro (bottom half) Levels.

phases. However, studies have shown that over 75% of project costs originate specifically from the maintenance phase [10, 115].

The bottom half of Fig. 1.1 shows the micro level activities. In contrast to the macro level, the recording of the data is automatic without additional effort. We envision that current maturity of tools and techniques of data available now at the micro level can be utilized to harmonize current SPI models, thus creating an additional dimension to the quantitative analysis of SPI.

With the rise of Open Source Software (OSS), there is also a need for SPI. Unlike conventional SPI models which are mainly controlled and driven by upper management, the livelihood and growth of OSS projects are greatly influenced at the micro level by developers. Therefore, we believe that the analysis at the micro level can assist with the adoption of SPI for OSS communities.

In this dissertation, we introduce Micro Process Analysis (MPA). This is defined by the use of data mined at the micro level: software repositories, reported

bugs and patches in issue tracking systems and peer reviews of code changes to reconstruct fine-grained processes of the daily activities of developers. This dissertation explores how MPA can be used to address the flaws of current SPI models. Since data collection is automatic, we assume higher integrity. Another advantage of using MPA is that, unlike typical SPI models, MPA allow us to focus on specific phases in the life-cycle. Additionally, since OSS is driven mainly by developers, MPA could be a suitable candidate to drive SPI initiatives.

To demonstrate the application of MPA, we targeted maintenance effort spent during maintenance phase of a software project. We applied MPA with two of the main software maintenance activities:

1. **Bug & Patch Processes** are micro processes related to the resolution of maintenance issues reported into the system. We proposed program slicing based metrics to study the relationship of maintenance effort in relation to the change impact on the source code.

2. **Peer Review Processes** are micro processes related to code inspection and review before merge of code change into the source code. Peer review member profiling for expertise identification can lead to a more efficient and higher quality review process.

This work has two main contributions. First, it provides an empirical approach for MPA, which can be used to supplement SPI models. We provide a quantitative proof of Lehman's second law of software evolution: *'As a system evolves its complexity increases unless work is done to maintain or reduce it'* [66]. (Part II). Second, we propose an approach for contributors to identify their social standing within an OSS project and visually map their pathway for career advancement (Part III).

1. Dissertation Organisation

This thesis is structured into three parts. The first part consists of two chapters. In the first chapter, brief backgrounds of micro process analysis, software quality metrics and the motivations are introduced. In the second chapter, we present a

literature review of the related fields to identify the current state of micro process analysis research. The results of the literary review is shown below:

- **Quantitative SPI of Software Maintenance** 44 papers were selected from 7 premium journals and conferences. 66% of SPI papers originated from the industry as opposed to only 10% from OSS and 7% SPI in other sources. This suggests that SPI has more industry contributions. However, only 7% of SPI are quantitative (correlation and experimental) studies. These results suggest that there is a gap between practitioners and researchers in the software process improvement (SPI). The review suggests there are opportunities for the use of micro process analysis to complement current models and metrics to provide quantitative SPI initiatives. (Chapter 2)

Part two has three chapters and presents the analysis of bug and patch processes at the micro level. In these chapters, we investigated the relationship between maintenance effort and its impact on source code. We propose an approach to quantitatively measure maintenance effort using code based program slicing metrics. In chapter four, we introduce the problem scenario, related literature and our proposed models and metrics. Chapter five and six discusses the application of our approach. Chapter five is the application of our approach with three open source projects while chapter six is performed in a controlled environment experiment. The results of the studies are shown below:

- **Micro Process Analysis of Maintenance Effort in OSS projects** We quantitatively examined the relationship of high maintenance effort and corresponding change impact of the code changes. In our case studies of three OSS projects, we determined that high level maintenance efforts also exhibited large change impacts on source code for project-specific processes. At statistically significant levels, results suggest the level of the maintenance efforts correlates with its impact on source code.(Chapter 5)

- **Experimental Study of Quantitative Analysis of Maintenance Effort** To eliminate outside factors influencing our proposed metrics for maintenance effort, we performed an experimental case study on a set of predefined maintenance activities. Our results suggested that program slicing

metrics have the strongest correlation with maintenance effort, exhibiting a moderate degree of correlation with maintenance effort. In contrast, the Lines of Code metric has a weak correlation with maintenance effort. This study contributes to our ongoing research into the analysis of maintenance processes.(Chapter 6)

Part three presents the analysis of micro processes of peer reviews. This part consists of three chapters, describing the use of micro process level metrics for profiling and understanding the career trajectories of peer review members of an OSS project. For this study, we solely use the Android project as our case study. In chapter seven, we introduce the Android peer review as well as our proposed peer review process models and metrics and related work. Chapter eight presents our profiling of peer review types while chapter nine extends the profiling metrics for use to mapping career paths of peer review types. The results of both studies are as follows:

- **Profiling Metrics to Categorise OSS Peer Review Types.** We investigated the three benefits of contributor profiling. First, the identification of hidden experts. Second, the identification of inactive or disinterested members to gauge the health of the OSS project. Finally, the assistance of aspiring members to monitor performance and identifying opportunities for career improvement. Preliminary results are promising, proving that our categories are practical, thus opening many avenues for future work. (Chapter 8)

- **Career Trajectory in an OSS Peer Review Community.** We proposed OSS historical career trajectory pathways for contributors. Results proved feasibility, opening many promising avenues for future work. Our study suggested that these models provide insights into their current standings in the project and based on historical evidence, possible career pathways for *career advancement*. (Chapter 9)

Our techniques and approaches proved the application of MPA to software maintenance activities such as bug fixing and peer reviews. The thesis finally concludes with a discussion and summary of contributions and an outlook into the future.

Part I

Micro Process Analysis (MPA)

Chapter 2

Background and Theory

1. Micro and Macro Process Levels

Osterweil first coined the use of macro and micro processes in software engineering [87]. He proposed macro process research to describe investigations that have emphasized the study of overall behaviours of process, while micro process research focused on the internal workings of the processes. It is not a new approach, in fact has been in use in fields such as economics, physics and the life sciences.

Nuseibeh further complemented with the idea of fine-grained software process modelling [86]. This was the idea of building software models at the developer levels. This was in contrast to the more coarse grained modelling concerned with more managerial and organisational activities. Morisaki and Iida then introduced micro process analysis as a study of the developer activity logs to extract the micro processes [79].

In this dissertation, we further explore the concept of using micro processes to assist in software maintenance process improvement. Additionally, with the emergence of mining software repositories (MSR) there are more opportunities to apply micro process analysis. Recent improvements of data repositories mining tools and techniques make this research timely as it enables quantitative insights.

2. Quantitative Software Maintenance Metrics

Software metrics are classified into three categories: product, process and project metrics [55]. As a subset of these metrics, software quality metrics focus on quality aspects. Software quality metrics are further broken down into three groups: product quality, in-process quality and maintenance quality. In this dissertation, we specifically focus on the product and process metrics concerned with maintenance quality.

It is very important to differentiate between maintenance and maintainability. According to the IEEE standard definitions, the following are definitions:

- **Maintenance:** The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

- **Maintainability:** The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.

In the growing field of software engineering this dissertation covers several aspects. Generally our work covers the field of Software Process Improvement (SPI), however since we are specifically interested in the maintenance phase we also incorporate the Software Maintenance (SM) field.

SPI is concerned with the improvement of software processes, and has a great number of international initiatives such as CMM and CMMI [61], SPICE (ISO/IEC15504) [34], ISO/IEC 12207 [1] , TSP [2] , GQM [16] and ISO 9000 [99]. These models have been the standard that mostly used by organisations to monitor and control their software processes. Most of these models were designed from a macro level perspective, thus most use only project metrics.

SM is more specific, relating to the processes and activities carried out during the maintenance phase of the software life-cycle. Consistent with the definitions, according to ISO/IEC 14764 the maintenance process can be divided into four types:

[1] http://www.12207.com/

[2] http://www.sei.cmu.edu/tsp/

- **Corrective Maintenance:** Maintenance performed to correct faults in hardware or software.
- **Adaptive Maintenance:** Software maintenance performed to make a computer program usable in a changed environment.
- **Perfective Maintenance:** Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.
- **Preventive maintenance:** Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

As mentioned in the previous chapter, it is well-known that the maintenance phase is the most costly in the software development lifecycle. In this dissertation, we focus on MPA of maintenance activities. In particular, we studied both the corrective maintenance (bug & patch) and the peer review process, which covers quality assurance of all maintenance activities.

3. Motivation for Study

In this section we provide a brief introduction and motivation to the two main parts of the dissertation. Section 3.1 refers to Part II and section 3.2 corresponding to Part III.

3.1 Supplements for SPI Models: Bug & Patch Processes

The assessment and improvement of software processes is rapidly gaining attention as an important activity in software development, with benefits seen in terms of cost-efficiency and improved business value [48]. More specifically, these benefits include improved productivity of development, and early defect detection and maintenance, which all account for a faster time to market [48].

Studies have shown that the maintenance phase of a software development lifecycle consumes a substantial amount of time and effort as compared to the other

development phases [10, 115]. Conventionally, to reduce these costs, most organisations employ Software Process Improvement (SPI) activities. However, more traditional process assessment models such as the Capability Maturity Model Integration (CMMI) [61] are generic and tend to cover all phases rather than focusing on a particular phase. To address this shortfall, in this part of the dissertation, we primarily focus on the analysis specifically at the maintenance phase.

Several studies, however, have pointed out some issues relating to current software process quality assessment methodologies such as CMMI [61] and international standards (i.e., ISO 9000) [99]. Most of these issues relate to the high costs of assessment and implementation.[105]. Hall and Baddoo both pointed out that these methodologies can be rather tedious, generic and complex as they assess all phases of the development life-cycle [43, 12]. In addition, studies have shown assessments to be higher management support, training, awareness, allocation of resources, staff involvement and experience of staff as de-motivators of software process assessments [93, 83]. Putting together all these factors, current software process assessment models pose difficulties in usage, especially for smaller software development organisations. Furthermore, process assessment includes other aspects such as process effort, human and infrastructure management and the achievement of process objectives.

To address the high costs and generic features of CMMI, work similar to Pino [90] suggest processes and models that focus on SPI for Very Small Entities (VSE - smaller organisations with less than 25 employees). These models are effective, however, as mentioned by Colla [30], much like SPICE, the focus is often at the macro level. Different to our approach, these models are driven at the higher level, therefore still suffer from the lack of quantitative analysis. We take a different approach by providing a quantitative method of supplementing these macro models.

The goal of this dissertation is to investigate a simpler and more accurate approaches for assessment of software process quality. Due to the complications of assessments of software quality across the entire software development life-cycle, this study investigates a much easier assessment approach, focusing primarily on the maintainability aspect of software product quality. Studies have shown

that the maintenance phase consumes a substantial amount of time and effort as compared to the other software phases during the software development life-cycle [10, 115].

The ISO/IEC 15504 Software Process Improvement and Capability Determination (SPICE) assessment model [34], does address some of the mentioned issues of CMMI. However, in contrast to our approach, SPICE and CMMI both have a higher level of abstraction. SPICE metric results are based on ratings (process attribute rating and attribute indicators ratings). However, we propose a much quantitative approach, which is more fine-grained than SPICE. As well as being cost-efficient, our approach provides results at a more fine-grained level than SPICE and CMMI, specifically for the quantitative assessment of the effects of maintenance efforts on source code maintainability.

Another aspect that is not present, purposely, in both SPICE and CMMI are detailed technical methods for process assessments. Conversely, our approach uses data-mining techniques and quantitative evaluations of the source code to assess processes, thus exploring the relationship between process and product.

Part II of the thesis introduces a novel approach measure maintenance effort and its impact on source code. Using three OSS projects and a closed experiment we proved this relationship to be statistically significant.

3.2 Adoption of SPI for OSS: Peer Review Processes

In a software project team it is assumed that the most skilled and knowledgeable members come through `experience`, defined by their length of membership *tenureship* and their historical activities. As an aspiring young member, the attainment of experience opens possibilities to more project responsibilities, career advancement and social standing among their peers. From a management standpoint, increases in demands/workload due to growth or losing key members could cause vacancies within the project. Management then has the tedious task of hiring or promoting current community members.

Visibility of the project team structure and individual identity is even more notorious in the emerging Open Source Software (OSS) project setting. In most instances, members are physically distributed, without face-to-face communication. In recent times, OSS projects have evolved into large and fairly complex

systems, arguable competing with their commercial counterparts.

Member's active participation and interest is vital to the livelihood of an OSS project. Bird et al. [23] states that the vitality of an OSS project depends on *"it's ability to attract, absorb and retain developers or face stagnancy and failure"*. Since OSS members are motivated purely by self-interest [64], visibility of ones standing and career paths could serve as further motivation for sustained activity. Advancements in data mining and tools and techniques have given birth to this emerging perspective, also investigated by Capiluppi [27]. Additionally, members may seek different career paths. For instance, some members may be content with being just moderate submitters or reviewers.

In part III of the dissertation, we propose a quantitative approach to profiling and career mapping models based on MPA of the peer review process.

4. Related Work

The dissertation covers a broad spectrum of related literature. Therefore, specific related work are presented in the earlier chapter for both parts of the dissertation. These can be found in both chapters 4 (bug & patch) and 7 (peer review).

5. Chapter Summary

In this chapter, we provided a background on the macro and micro process levels of software development . Since this dissertation is concerned with quantitative metrics we highlighted the different types of quantitative software metric types. Later, we introduced the models used in SPI and the different types of SM. Finally, we introduced the motivations for the two types for the bug & patches as well as the peer review micro processes.

Chapter 3

Systematic Literature Review of SPI and SM

1. Motivation

This chapter presents a systematic literature review (SLR) of studies related to fields of Software Process Improvement(SPI) and Software Maintenance(SM). Since the scope of this dissertation covers two fields of Software Engineering, the objective of the study is to analyse the trends of each field and how they interact/complement with each other.

2. Review Process

Following the guidelines from Kitchenham et al. [59], we adopted the following systematic review steps:

- Step 1 - **Plan Review.** We define our research questions, publication selection and criteria and analysis design.
- Step 2 - **Conduct Review.** Data extraction, filter using criteria and classification of publications.
- Step 3 - **Analysis.** Discussion, consider threats and draw conclusions.

Table 3.1: Targeted journals and conferences

Name	Year
IEEE Transactions on Software Engineering (TSE)	2002-2011
Journal of Software Maintenance and Evolution (JSME)	2002-2011
Software Process: Improvement and Practice (SPIP)	2002-2009
International Conference on Software Engineering (ICSE)	2002-2011
International Conference on Software Maintenance (ICSM)	2002-2011
International Conference on Software Processes (ICSP)	2005-2011
Working Conference on Mining Software Repositories (MSR)	2004-2011

2.1 Plan Review

Research Questions

Based on the motivations outlined in the previous chapter, we constructed the following research questions:

- **RQ1:** What are the main data sources of the surveyed papers?
- **RQ2:** How many are quantitative studies of the maintenance processes at the micro-level?

Paper Selection Criteria

Since Kitchenham et al. [59] reported that automatic search methods are prone to quality issues, we decided to instead perform a manual search. As shown in Table 3.1 [1] [2] [3] [4], we targeted the premium conferences and journals related to SPI and SM.

The selection criteria are outlined below:

[1] As of 2012, JSME is now integrated with Journal of Software: Evolution and Process

[2] Since 2009, SPIP was integrated into now the Journal of Software: Evolution and Process

[3] Since 2008, ICSP has changed its name to the International Conference on Systems and Software Processes (ICSSP)

[4] Before 2008, MSR was originally a Workshop

IC1: The title or keywords includes the following terms; *Software Process Improvement, maintenance efforts, mining, bugs, micro processes, fine-grained processes.*
IC2: The abstract contains sufficient information relevant to the three fields.
Additionally, we enforced the following exclusion criteria for quality purposes:
EC1: For SPI related papers, the paper either focuses on smaller organisations or de-motivation of current SPI initiatives.
EC2: For SM related papers, papers that investigate maintenance efforts and mining techniques of bugs and maintenance related activities.
EC3: The papers intersect either field.

To reduce potential individual bias and for future replication, we propose to employ a group critic session. The group will consist of knowledgeable individuals in software engineering concepts. If a paper is deemed inappropriate, then the full paper would be consulted for re-evaluation.

Analysis Design

Based on all research questions, we first needed to classify the collected papers according to their respective fields of either SPI or SM. Using the order of the keywords, we can classify each paper. To address R1, we classify papers on the data source, as being either from the industry, open source project or other sources such as theory, experimental conditions or other public accessible sources such as the PROMISE repository [5] . For R2, we evaluate the type of research performed. This is discussed in detail later in the chapter.

2.2 Conduct Review

Using the inclusion criteria, we manually collected 290 papers mainly by extracting the title and abstract from either the journal/conference websites, the ACM

[5] http://promise.site.uottawa.ca/SERepository/

digital library [6] , IEEE Xplore [7] and the DBLP online database [8] .

Then using the exclusion criteria, we systematically removed inappropriate and redundant papers. Finally, the group critic session validated the quality of the selected papers. Our final results contained 44 articles (14 journals and 30 conference/workshop papers).

2.3 Analysis

To assist in the study of the context and the nature of each paper, we prepared the following both general and field-specific classifications. The general classification are adopted from Creswell[31]:

- **Comparative (Qualitative)** - This research is a comparison of two models or approaches, discussing the benefits and flaws.
- **Correlation(Quantitative)** - These papers are 'analytical surveys' describing a statistical measure of relationship between two phenomena.
- **Descriptive (Qualitative)** - In this study, these refer to papers that are an examination of a standard model or theory.
- **Evaluation (Qualitative)** - These studies are meaningful constructions of complex social, cultural and psychological issues.
- **Experimentation (Quantitative)** - These papers attempt to isolate and control all relevant conditions, so as to observe the effects of when conditions are manipulated.
- **Action Research (Qualitative)** - Similar to experimentation papers but in a real world setting.

As shown in Table 3.2, we designed field-specific classifications. These classifications will be used to address RQ1 and RQ2. The overall goal of the classifications were to distinguish the context, sources, and types of studies related to

[6] http://dl.acm.org/

[7] http://ieeexplore.ieee.org/

[8] http://www.informatik.uni-trier.de/ ley/db/index.html

Table 3.2: Field specific classifications

Field	Name	Classification Indicators
SPI	Model	Analysis of new, tailored or classic SPI models
	Benchmark	Comparison of SPI intiatives across projects
	Experiential	Document experiences of SPI initiatives
SM	Effort	Analysis of maintenance efforts
	Experiential	Document experiences relating to software maintenance
	MPA	Analysis of maintenance activities at the micro-level
	MSR	Use of mining software repository techniques

this dissertation.

2.4 Threats to Validity

We identified both potential internal and external threats to the validity of the systematic review and its results.

Internal

There is a risk of publication and researcher bias in this review. Publication bias refers to the general problem that positive research outcomes are more likely to be published than negative ones [59]. We do not regard this as a threat, especially since our SPI related papers focus on shortfalls of classical SPI models. For researcher bias, as mentioned in our analysis design, we have consultants to validate and ensure the process is repeatable and reliable.

External

To address the generalization of the results, we selected papers from premium journals and the most well-known, highest ranking conferences related to both SPI and SM in the software engineering field. We are confident that the results are illustrative of the current state-of-the-art.

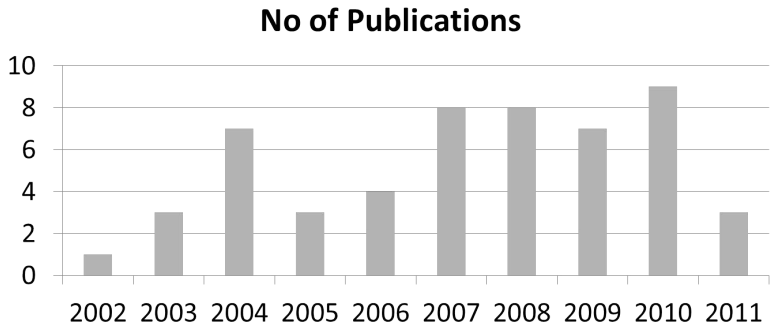


Figure 3.1: Number of publications per year

3. Results

3.1 Specialised Field

Fig. 3.1 shows the number of publications per year from 2002 to 2011. From 2002 to 2006, there is a steady rise in papers published, with a leap in 2004. This spike could be explained by the establishment of the MSR working conference, recognising the field of mining repositories for analysis at the micro-level. Since 2007, the number of papers have steadily increased until 2010.

Taking a closer look, we see in Fig. 3.2 the publications by their respective fields. Generally there has been more research with SPI until 2010, where we see a drastic drop together with an increase on SM papers. From 2007 and onwards, research in both fields seem to increase. This could also be due to the improvement of respository tools such as CVS, SVN and increased email communication, enabling this research. From 2008, SPI papers have decreased. This could be accounted for the merge of SPIP journal into the JSME journal in 2009.

3.2 Sources of Studies (RQ1)

Fig. 3.3(a) shows the publications per year by data source. Since 2004 and then 2007 onwards, there has been an increase on other sources of data, compared to industry data. According to Hata [47], this is consistent with previous studies that suggest that there was an increase in public datasets such as the PROMISE

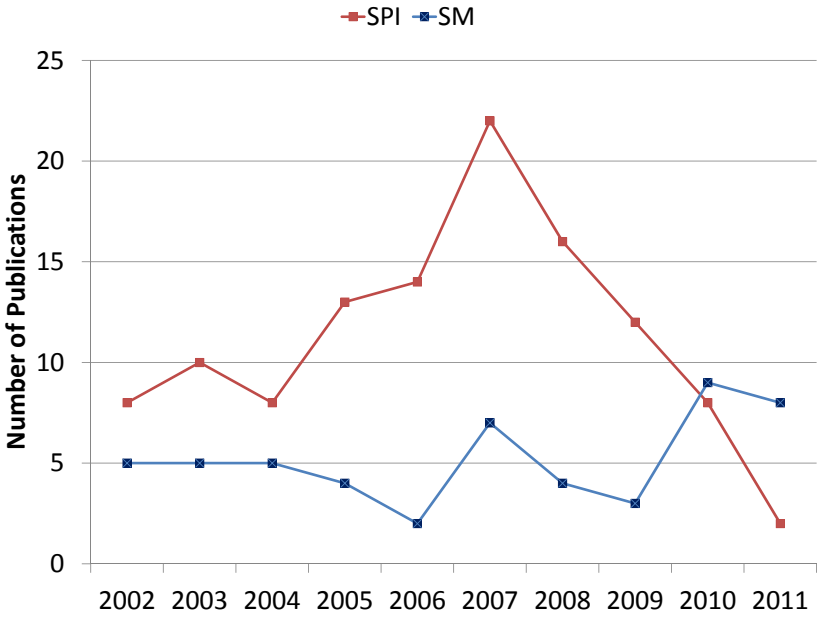


Figure 3.2: Field of study per year

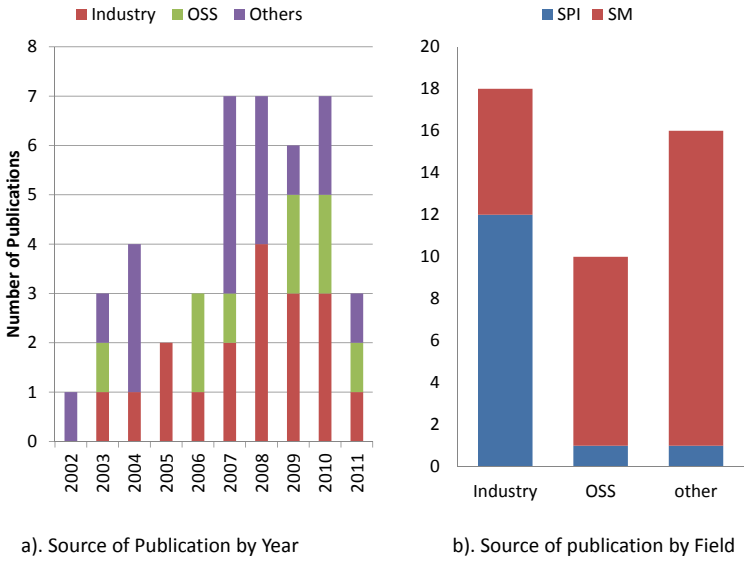


Figure 3.3: Sources of publications

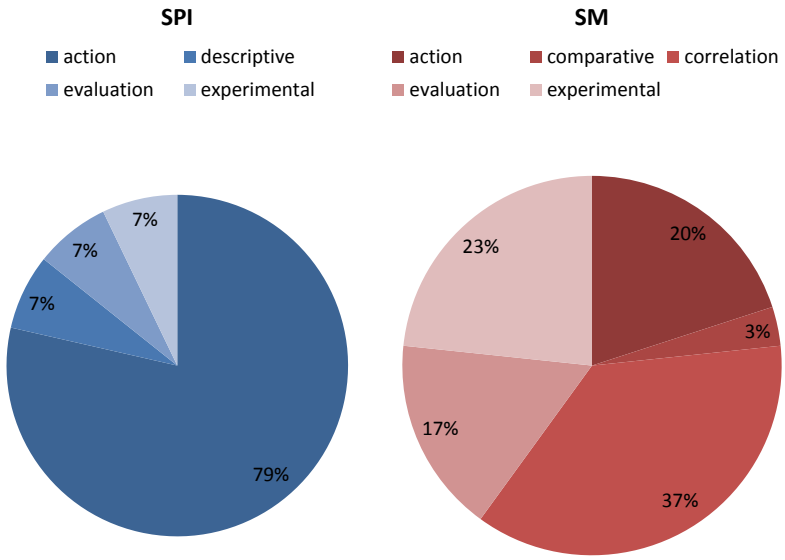


Figure 3.4: Research methods by field

repository.

As seen in Fig. 3.3(b), we further classified the data sources by specialized field. Results suggest that most SPI studies originate from the industry (66%). In contrary, most SM studies come from other sources (93%) and OSS (90%).

3.3 Research Methods (RQ2)

Fig. 3.4 illustrates the breakdown of the types of studies undertaken in the selected papers. Since most the SPI papers originate from the industry, it is not surprising that 79% of the papers are qualitative (action research) papers. On the contrary, SM papers (37% correlation and 23% experimental) are quantitative experimental research. These results address RQ2.

3.4 Classifications by Study Type (Additional)

Table. 3.3 and Fig. 3.5 show the results of the classification employed in this review. In Fig. 3.5, it is shown that 57% of the selected SPI papers were benchmark studies, followed by models (29%) and then experiential (14%). Over half (53%)

Table 3.3: Classification of papers

Field	Classification	Papers
SPI	Model	[29, 119, 91, 107]
	Benchmark	[84, 13, 98, 108, 56, 111, 106, 67]
	Experiential	[71, 112]
SM	Effort	[85, 75, 54, 78, 117]
	Experiential	[52, 103, 101, 3, 2]
	MPA	[7, 15, 51, 120]
	MSR	[114, 73, 19, 18, 53, 17, 8, 62, 39, 82, 41, 92, 42, 6, 96, 114]

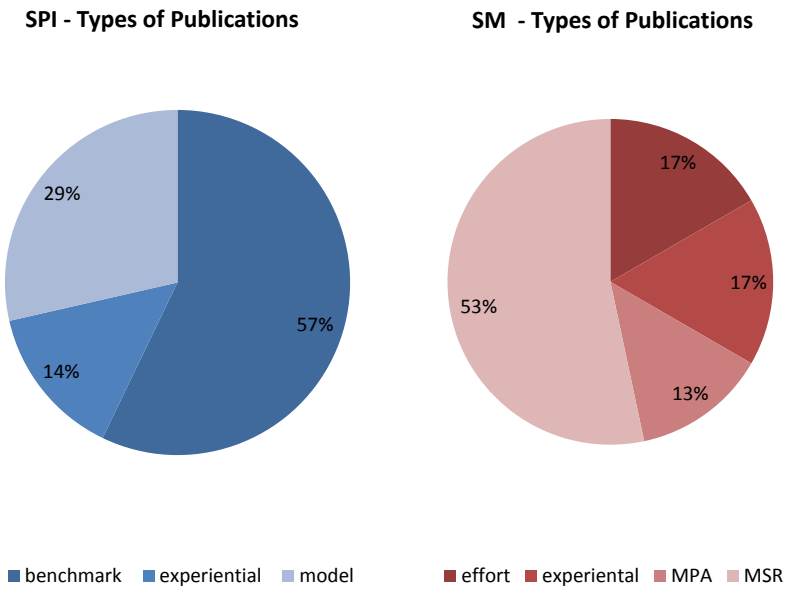


Figure 3.5: Classification by field

of the SM papers are mining repositories (MSR) related. Other were experiential (17%), effort (17%) and MPA (13%) related.

4. Chapter Summary

We conducted a systematic review from both the SPI and SM fields. Papers were selected from three journals and four conferences. From our results, we can now answer the research questions.

RQ1: *What are the main data sources of the surveyed papers?*

According to our results, most SPI research has originated from the industry. On the contrary, SM research has come from OSS and other sources. The results suggest that there could be a difference in representation between the practitioners and researchers in each respective field. Therefore, our research focused on SPI for OSS projects should contribute to bridging this gap.

RQ2: *How many are quantitative studies of the maintenance processes at the micro-level?*

Based on the data in Fig. 3.5, the correlation and experimental study types account for 43% of all studies. However, only 7% are SPI related. Together with RQ1, the results suggest that most industrial studies of SPI are not quantitative. Therefore, our studies suggested that there is a gap in quantitative for SPI. Specifically, as shown in Table 3.2, out of the 44 studies, only 4 were MPA related.

To conclude, this literature review suggests that there is a gap for quantitative SPI studies. We envision that our MPA research for OSS projects are the micro level and should have significant contributions to both SPI and SM fields.

Part II

Bug & Patch Processes

Chapter 4

Bug & Patch Process: Model and Metrics

1. Introduction

In this chapter, we investigate the maintenance effort in relation to its change impact on the source code. As a quantitative proof of Lehman's law, we proved this phenomena at statistically significant levels.

2. Background and Related Work

The related literature covers a wide range of fields, however, each has slightly different motivations and approaches. We divided the related works into detection and prediction metrics, program slicing, and change impact analysis. To the best of our knowledge, our objective is novel as we propose techniques to quantitatively assess software processes.

2.1 Software Process Analysis and Assessment

Several related works have attempted to address the shortfalls of current software process assessment models. Yoo [116] suggested a model that combined CMMI and ISO models. Armbrust [9] took a different approach by treating software as manufacturing product lines, creating easier processes, however, making the

processes systematic and generic. Unlike these approaches, we use the data mined from software repositories for our assessment.

2.2 Detection and prediction metrics using software repositories

Most work related to mining repositories has had objectives related to detection and prediction of fault proneness in the functions, modules and features during maintenance [45, 57]. Zimmerman [68] used the extracted information to measure against software patterns. Fisher [36] mined bug reports and version control systems, using visualization techniques to understand features.

Like the maintenance effort in this study, similar work studied the effort spent to fix bugs [110]. Weiss and colleagues estimated the effort to fix an issue based on prior similar issues, applying time as an indicator for effort. Also, work by Kim [58] referred to the time to fix bugs as an important factor. Other related works in this area have proposed heuristic approaches to measure the impact of code changes [121, 113]. As compared to Kim, we introduce the complexity of the micro processes to further express maintenance effort.

Models commonly use product metrics for analysis. However, Kamei et. al [54] proved that process metrics outperform product metrics for bug prediction models. Our work uses combination of process (complexity of the micro processes) and product metrics (our proposed program slicing based metrics) for our analysis. We explore the process-product relationship, similar to older models such as the PROFES (PROduct Focused improvement of Embedded Software processes) improvement methodology [20].

2.3 Program Slicing

First proposed by Weiser [109], Program Slicing refers to a subset of a program's behaviour, reducing a program to its minimal form which still produces that behaviour. Based on a slicing criterion, program slicing can isolate interprocedural dependencies at either the module, file or function level.

Many of the metrics widely used in the field of program slicing are related to the evolution of code [44, 74]; among those, many are cohesion and coupling

based approaches. Similar research has used program slicing metrics to classify bugs using these metrics [88, 63]. Instead of the standard slicing metrics, our proposed metrics include a count of the number of functions affected to measure size and the cyclomatic complexity of the code to measure complexity within the slices.

Work by Nagappan's group is very similar to ours, but, with a different objective. They evaluated Windows Server 2003, and assessed the relationships between the software dependencies and churn measures with the objective of finding efficient predictors of post-release defects [81]. Our work has the objective of assessing software processes.

2.4 Change Impact Analysis

Program Slicing is well known in the field of change impact analysis. Gallager [38] illustrated its usefulness as it assisted program comprehension, more specifically guiding developers to determine which code components were not related to a software change. Similar to this, Differential Symbolic Execution (DSE) characterized the effects of a set of program changes in terms of behavioural program differences [89]. There also has been research to predict if a software change is clean or buggy [57]. Canfora applied program slicing as well, to indexing changes [26]. German and Hassan and Robles explored the use of change impact graphs to visualize the impact of code changes to investigate real defects [40]. Hassan predicted faults using the complexity of code changes [46]. Much like our research, using information theory, Hassan deduced that code changes with complex micro processes negatively affect a program. Unlike these other efforts we also have a different motivation and objective.

3. Issue Resolution Model

Fig. 4.1 illustrates the base micro process model used in the studies. The model consists of three steps. In the first step, the issue is detected and reported into the system as a *new* state. In the second step, the issue undergoes various states until it is resolved. For instance, most issues usually change its state to either *confirmed* and/or *accepted* before developers begin committing code changes. The second

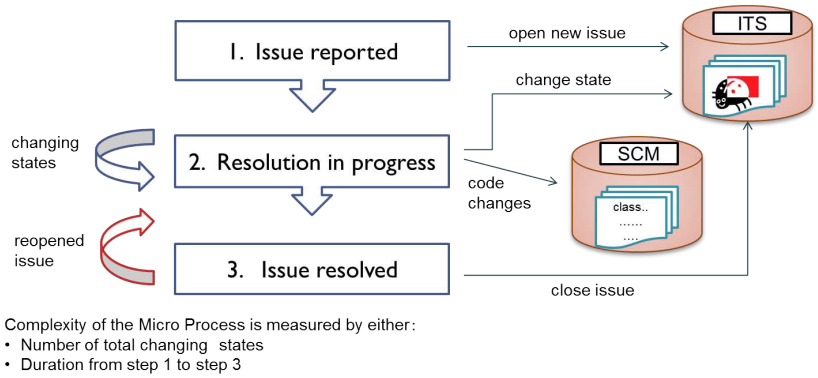


Figure 4.1: Micro process analysis model

step is concluded once all code changes needed to resolve the issue are committed to the source code. Moving into the third step, the issue changes its state to *closed*, thus marking the issue as being resolved. There are some cases when the solution is not sufficient, so the issue changes its state to *reopened*, reverting the micro process to the second step.

4. Proposed Metrics

To measure the impact of the maintenance effort, we propose metrics based on the behavioural properties of the program. We applied two metrics, 1). McCabe's Cyclomatic Complexity (CC) to measure the *complexity* of the changes and 2). Function Count (FC) to measure the *size* of the changes. We selected these two parameters as they are two widely accepted and relatively simple analytical metrics [35]. In addition, we introduced non-program slicing counterparts. Equations (4.1, 4.4) to evaluate the effectiveness of using the program slicing technique.

Fig. 4.2 illustrates how our approach applied program slicing. For each issue, we assume that each file edited during a code change is stored in the SCM as a revision. Therefore for each affected revision we identify all the functions modified during the code change. We refer to these functions as *edited functions*. For every edited function, we then calculate the *backward slices* and the *forward slices*. The backward slice is the set of functions that the edited function depends on. The forward slice refers to the set of functions that depend on the edited function.

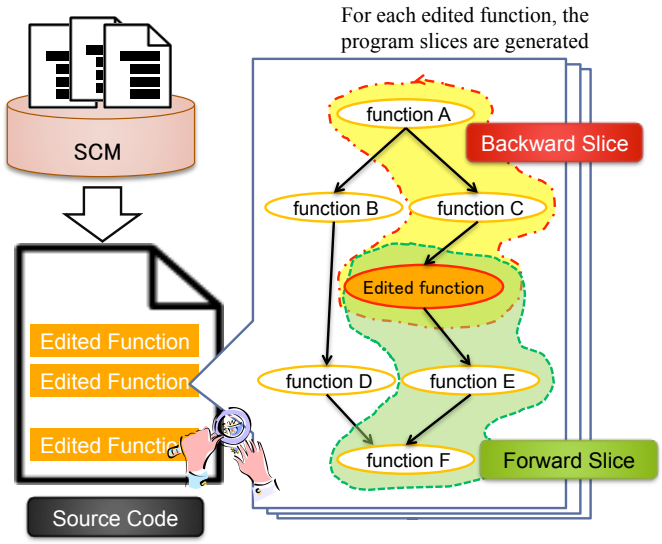


Figure 4.2: Example of *backward slice* and *forward slice* for a program slice of an edited function.

Program slicing ensures that only the source code related to the edited functions is analysed.

Formally, we define code changes as a sequence of revisions $\mathfrak{R} = \langle r_1, r_2, \cdots \rangle$. For each revision r, we define $\mathbb{F}_{E,r} = \{f_1, f_2, \cdots\}$ as the set of edited functions in revision r. Given the edited function f , edited in revision r, we define $S_B(f)$ and $S_F(f)$ as backward and forward slices of f. Note that we can assume that $S_B(f)$ is the set of functions which affect f and $S_F(f)$ is the set of functions affected by f.

To define the following metrics, we introduce $\mathbb{S}_{B,r}$ as the backward slice and $\mathbb{S}_{F,r}$ as the forward slice in a revision r. Trivially, $\mathbb{S}_{B,r} = \bigcup_{f \in \mathbb{F}_{E,r}} S_B(f)$ and $\mathbb{S}_{F,r} = \bigcup_{f \in \mathbb{F}_{E,r}} S_F(f)$.

4.1 Cyclomatic Complexity (CC) based Metrics

Given a function f, we define the function $C(f)$ which gets the CC of f. These proposed metrics shown in Equations (4.1, 4.2, 4.3) are used to measure the total complexity of all functions in the slice at a certain revision.

- **EditedFunctionCC.** This is the summation of the CC for functions edited during a code change, i.e.,

$$\sum_{f \in \mathbb{F}_{E,r}} C(f). \tag{4.1}$$

Rationale: This is the non-program slicing metric for comparison against both Equations (4.2, 4.3).

- **BackwardSliceFunctionCC.** This is the summation of the CC for each function in $\mathbb{S}_{B,r}$, i.e.,

$$\sum_{f \in \mathbb{S}_{B,r}} C(f). \tag{4.2}$$

Rationale: This metric computes the combined total CC of the functions that $\mathbb{F}_{E,r}$ is dependent on.

- **ForwardSliceFunctionCC.** This is the summation of the CC for each function in $\mathbb{S}_{F,r}$, i.e.,

$$\sum_{f \in \mathbb{S}_{F,r}} C(f). \tag{4.3}$$

Rationale: This metric computes combined total CC of the functions depending on functions in $\mathbb{F}_{E,r}$.

4.2 Function Count (FC) based Metrics

To measure the size of the code change, we introduce three metrics shown in Equations (4.4, 4.5, 4.6) based on the number of functions affected by the code change in revision r.

- **EditedFC.** The number of functions edited during a code change, i.e.,

$$|\mathbb{F}_{E,r}|. \tag{4.4}$$

Rationale: This is the non-program slicing metric for comparison against both Equations (4.5, 4.6).

- **BackwardSliceFC.** The number of functions in $\mathbb{S}_{B,r}$, i.e.,

$$|\mathbb{S}_{B,r}|. \tag{4.5}$$

  Rationale: This metric computes the number of functions that $\mathbb{F}_{E,r}$ is dependent on.

- **ForwardSliceFC.** The number of functions in $\mathbb{S}_{F,r}$, i.e.,

$$|\mathbb{S}_{F,r}|. \tag{4.6}$$

  Rationale: This metric computes the number of functions depending on functions in $\mathbb{F}_{E,r}$.

5. Contributions

Part II (Chapters 5 and 6) of the dissertation makes the following contributions:

New approach to the assessment of software processes. At a fine-grain level, our approach focuses on the maintainability effort and its relation to the code.

Quantitative expression of maintenance effort in terms of micro processes. Using project-specific workflows to resolve issues, we used the complexity and duration of the micro processes to quantitatively calculate maintenance effort.

Proposed program slicing-based metrics to measure change impact. We introduced four program slicing-based metrics at a more precise function level, measuring the complexity (based on McCabe's Cyclomatic Complexity) and size (the number of functions sliced) of an issue.

Determine high maintenance efforts. We were able to identify high maintenance efforts based on the distribution of all maintenance efforts within the software project. We discovered that high maintenance efforts usually have high change impact on the source code.

Statistically significant correlations between the maintenance efforts and its change impact on source code. Using the standard t-test, our proposed metrics improved its p-values over most of the corresponding non-program slicing metrics.

Application to different projects. Our approach yielded similar results across projects that differed in source code size, workflow, data management systems and the handling of issues during the maintenance phase.

Quantitative correlations of maintenance effort to our proposed metrics. Our program slicing-based metrics showed moderate to strong (ρ = 0.7-0.8) degree of correlation. In contrast LoC was shown to have a very weak correlation (ρ = 0.35) with maintenance effort.

6. Chapter Summary

In this chapter, we introduce the motivation and related work for the study of bug and patch processes at the micro level. We also explain in detail our proposed models and metrics used for our research with contributions. Case studies are introduced in Chapters 5 and 6.

Chapter 5

Case Study: An OSS Setting

1. Introduction

In Chapter 4, we introduced our proposed model and metrics for MPA of bug and patch fixing processes. In this chapter, we demonstrate feasibility in three open source software projects. We demonstrate that maintenance effort affects the quality of the source code, which pertains to its maintainability. We suspect that for each project, we can determine the high maintenance efforts and relate them to having high impact on the code. To test our assumptions, we constructed the following research questions:

- ***RQ1.*** *Are we able to determine high maintenance effort?*
- ***RQ2.*** *How much impact does a high maintenance effort have on the source code?*
- ***RQ3.*** *Is there a correlation between maintenance effort and its impact to source code?*

To evaluate our research questions and proposed approach, we performed a case study of three open source projects. Mining data from a source code repository and issue tracking system, we were able to measure the maintenance effort (based on micro process analysis) and change impact (based on the proposed metrics). Since most micro processes follow a tailored workflow, we developed

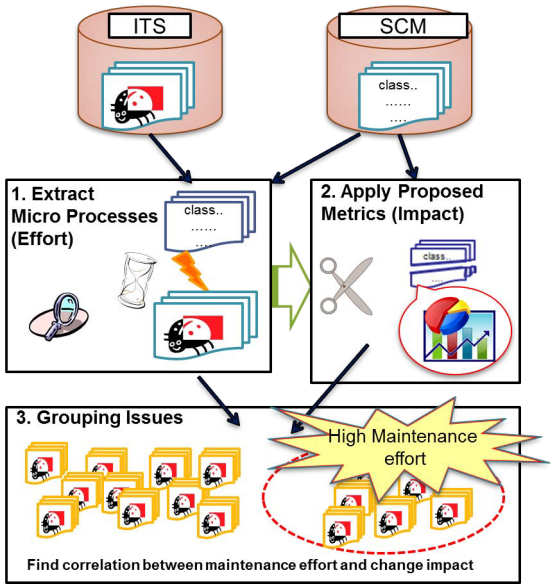


Figure 5.1: An Overview of our approach

thresholds to determine project-specific high maintenance efforts. Results indicated that maintenance effort had statistically significant correlations with the impact on the source code.

Our approach offers a supplement approach for SPI assessment at the micro level. Based on project-tailored micro process analysis, our approach can be used to help determine where and how maintenance efforts can be reduced (i.e., the proper assignment of resources) and if the affected portions of source code are candidates for maintenance activities such as refactoring, code inspections and reviews.

Our maintenance effort thresholds enable our approach to be specifically tailored to any project. Since only three projects are used in this study, future replication across a wider range of projects is needed to generalize our approach.

2. Approach

2.1 Proposed Approach

Given this background, we proposed a three-step approach as shown in Fig. 5.1. The first step is the extraction of the micro processes from both the SCM (Source Code Management) and the ITS (Issue Tracking Systems). The extraction provides sufficient data to reconstruct the micro processes of the maintenance effort. In the second step, we introduce the proposed program slicing metrics to identify impact in relation to the maintenance effort. In the third step, we propose grouping parameters to evaluate each maintenance effort. Based on the micro processes, we determine and group the high maintenance efforts. These steps are explained below.

Step 1. Extraction of Micro Processes.

Our goal of mining the software repositories was to extract sufficient information for each issue to be able to reconstruct the related micro processes, and measure the maintenance for an issue. The micro process involves all the processes from when an issue is first opened until it is closed.

Fig. 5.2 shows screenshots of the software repositories of an issue that has been resolved. The example shows a typical change log, with an issue stored using the TRAC ITS and a source code revision retrieved from an SVN (SubVersioN) SCM system. To reconstruct this micro process, the highlighted information needs to be extracted from the software repositories. In addition, Table 5.1 contains a description of the data extracted that is needed for our approach. We specifically designed our tool to extract data from the TRAC ITS and SVN systems.

As seen in Fig. 5.2, the extraction method is based on the identification of the linkage between the issue and the revision in which the code changes were committed. Our extraction method is based on the two main approaches used in the field. The two well-known methods for extraction of bug-fix data are by Fisher et. al [37] and Chen et. al. [28]. The first involves searching the change logs for keywords such as '*bug*' or '*fix*' to extract the bug data while the other manually compares the correctness of change logs. In this research, we applied a combination of both methods. Additionally, we chose a project that had been

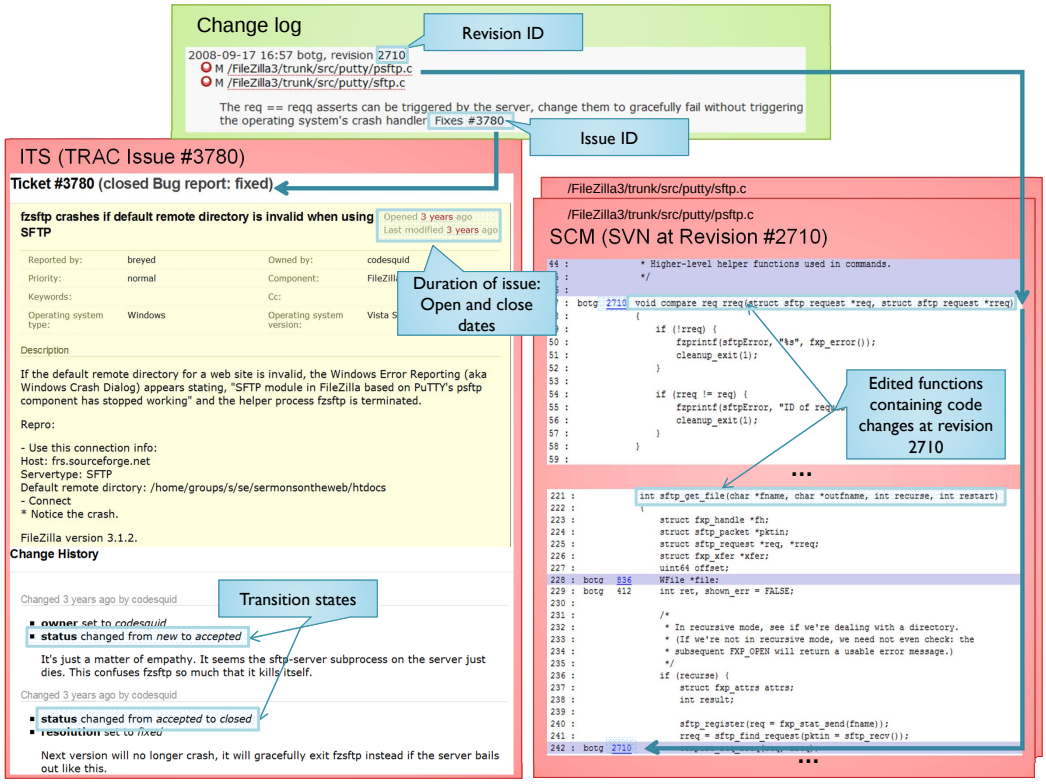


Figure 5.2: Example illustrating issue 3780 and corresponding rev. 2710. This shows the data needed to reconstruct the micro process (issue report) and related code change impact(source code)

studied previously, which was known to have high quality change logs. Our approach searches for code changes based on keywords such as '*bug*' or '*fix*' as well as linkages (i.e., referred to as either bug ID or issue ID) from within change logs of the source code repository. As seen in Fig. 5.2, extraction of the functions is done by comparing the changes to the previous revision. We implemented this methodology using our data mining tool. This tool combined simple web scripts to download, parse and extract the required data from online open source repositories into a relational database for analysis.

Table 5.1: Information extracted from the software repositories

Software Repository	Attribute	Description
Issue Management System	Issue ID (Bug ID)	Identification of the issue
	Date Open	Date when the issues was reported into the system
	Date Closed	Date issues was resolved
	State	Transition states of issue
Source Code Repository	Revision ID	Identification to track changes made to source code
	Issue ID (Bug ID)	link issue to code change
	Edit date	Refers to the date when the latest code change was performed
	Edited functions	Function(s) where code was edited (compared to previous revision)

Step 2. Proposed Metrics to Measure Change Impact.

We use the proposed metrics in Chapter 4 to measure the change impact. To generate our metrics, the software analysis tool by GrammaTech called CodeSurfer [5], a sophisticated and widely used tool for interprocedural program slicing, was used [88, 4, 74, 22]. Customized scripts within CodeSurfer were used to calculate the CC and FC within the slices.

Step 3. Grouping Code Changes.

Grouping of issues based on their maintenance effort involves two steps: first is *pre-processing* to improve the quality of the data (known as cleansing), and second is *calculating the effort threshold* to determine and group high maintenance efforts. Once the threshold is defined, evaluations on the two groupings to test correlations are performed.

Pre-processing procedures (Cleansing of the Data).

To improve the quality of our results, we applied filters to the datasets in the experiment. This process removed data that could negatively affect the results and gave confidence in the quality of the data collected. From our collected code changes, we removed issues having the following criteria:

- *Open issues*: Code changes related to issues not yet resolved (i.e., not closed status)
- *Non-code related changes*: Code changes related to documentation or images (`changelog.txt, pic.jpg`...etc).
- *Build related issues*: Code changes related to compiling or build errors.
- *Revisions with no linkages*: Code changes with missing issue tracking information (i.e, no Bug ID references in the revision change log and vice versa).

Pre-processing was a two step process: 1) Our tool parsed the data extracted from the software repositories to remove code changes meeting the criteria mentioned above. To remove false positives, we then 2) manually checked all remaining code changes against these filters to ensure high quality results. Using the above criteria, all the cleansing of the data could be performed automatically, however, due to time constraints and the current limitations of our tool, the current implementation provides a semi-automatic verification of the datasets.

Grouping using effort thresholds.

Referring back to Fig. 4.1, we propose that the complexity of the micro processes for each issue contributes to its maintenance effort. This implies that more complex issues usually have more state changes or takes more time to resolve. Each micro process fragment is defined as the *state change* for an issue related to a set of code changes. For instance, an issue has a '*new*' state when first reported and finally ends up with a '*closed*' state when resolved. *Duration*, which is measured in the number of days, is defined as the time from when the code change is first requested in the issue change log (has a '*new*' state), until the time when the issue is closed ('*closed*' state).

Maintenance effort depends on both the *complexity* of the micro process as well as the *duration* until the resolution of the issue. As seen in Fig 4.1, the complexity of the micro-process can be measured by the combination of either the number of state changes and/or the duration of the issue before it was resolved. It seems likely that there is a mutual dependency between complexity and duration. For

instance, we are able to differentiate complex issues that were quickly resolved. We suggest that such issues could be prone to be reopened.

We formulated *effort thresholds*, designed to determine maintenance efforts that require more than the 'normal' state changes and duration. Formally, given an issue i, $S(i)$ is the number of state changes, and $D(i)$ is the duration. We introduced two effort thresholds: $\mathfrak{S}$ to represent the state change threshold and $\mathfrak{D}$ for the duration threshold. Analysis of the distribution of $S(i)$ and $D(i)$, allows us to identify the $\mathfrak{S}$ and $\mathfrak{D}$ thresholds specific for any project. We propose to set the effort thresholds based on the outliers of these distributions.

Based on the thresholds, we divided the issues into two groupings, those that require more than the 'normal' maintenance effort (higher effort needed to resolve denoted as $\mathbb{ME}_{\text{High}}$) and the rest of the issues (requiring normal or less effort to resolve denoted as $\mathbb{ME}_{\text{Normal}}$). *High maintenance efforts* refers to issues identified as being higher/above the effort thresholds. *Normal maintenance efforts* refers to issues identified as being less than/below the effort thresholds.

$$\mathbb{ME}_{\text{High}} = \{i | i \in \mathbb{ME} \wedge (S(i) \geq \mathfrak{S} \vee \mathfrak{D}(\mathfrak{i}) \geq \mathfrak{D})\} \tag{5.1}$$

$$\mathbb{ME}_{\text{Normal}} = \{i | i \in \mathbb{ME} \wedge (S(i) < \mathfrak{S} \wedge \mathfrak{D}(\mathfrak{i}) < \mathfrak{D})\} \tag{5.2}$$

Formally, given the *maintenance effort* $\mathbb{ME}$ of any issue , we define high maintenance efforts as Equation (5.1) and normal maintenance effort as Equation (5.2).

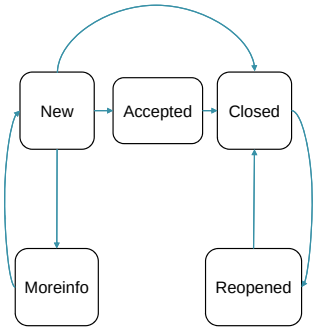
3. Case Study and Results

3.1 Experiment Setup

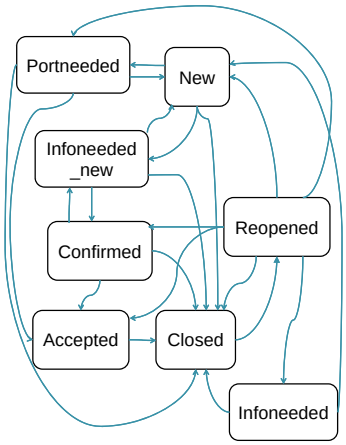
To test our approach, we chose three Open Source Software (OSS) projects for analysis. The chosen projects are Filezilla, which is a File Transfer Protocol application, WxWidgets, a C++ library that lets developers create GUI applications for major OS as well as mobile OS and embedded GTK+ architectures, and Lighttpd, a lightweight open-source web server. At the time when the study was conducted, the latest release of Filezilla (up to Ver. 3.3.1) had 210,629 Lines Of Code (LoC), WxWidgets, the largest (up to Ver. 2.9.0) had 409,148 LoC, while Lighttpd, the smallest (up to Version 1.5.0) had 40,712 LoC.

Table 5.2: Software Project Overview

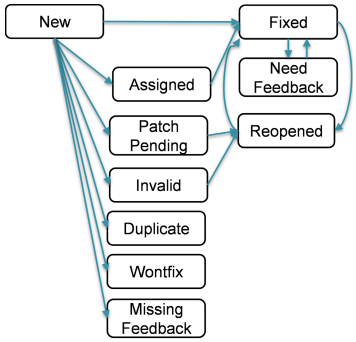
Project	Code Change Sets (after pre-processing)	Number of Functions	Time Window	Release
Filezilla 1	156(100)	8959	Aug 07-Jul 08	Ver. 3.1
Filezilla 2	173(136)	10488	Jan 09-Dec 09	Ver 3.3
WxWidget 1	358(304)	17836	Dec 06-Nov 07	Ver 2.6.4
WxWidget 2	347(303)	23203	Mar 07-Feb 08	Ver 2.8.0
Lighttpd 1	121(58)	856	Aug 07-Sept 08	Ver 1.4.21
Lighttpd 2	101(83)	840	Mar 05-Aug 10	Ver 1.4.28



(a) Filezilla workflow



(b) WxWidgets workflow



(c) Lighttpd workflow

Figure 5.3: Workflow for all projects

For the experiment, we randomly selected two versions from each project as datasets for analysis. These datasets, as shown in Table 5.2, are the basis on which the program slicing based metrics were calculated. Then, using the code changes from issue reports from the ITS, we calculated the proposed metrics of the code related to these versions. The time window shows the duration period in which the issues were resolved. Note that Lighttpd generally has a longer time window, especially Lighttpd Dataset 2 with almost 4 years to resolve an issue.

Both Filezilla and WxWidgets use the TRAC Management System [1] , while Lighttpd uses Redmine [2] as their ITS. Fig. 5.3 shows the workflow of each project. For Filezilla and WxWidgets, the workflows were constructed based on the workflow guidelines available at each TRAC repository. As shown in Fig. 5.3, WxWidgets has a more complicated workflow in comparison to Filezilla. Offering more choices for the developers, the WxWidget workflow offers more guidance. However, its implementation is questionable. On the other hand, Filezilla has a more simplistic approach. Therefore, in the case of Filezilla, the duration of an issue rather than the complexity of the micro process could be a better indicator of maintenance effort. Unlike the other two, Lighttpd had no documented workflow. Instead, as shown in Fig. 5.3c, it had a set of selectable states. Using the extracted issues we were able to construct a simple workflow.

For WxWidgets, we extracted data from the WxWidgets TRAC system (ITS) [3] and WxWidgets SCM [4] inspecting 64,005 revision changes. For Filezila, we extracted 3,611 revision changes from the Filezilla TRAC (ITS) [5] and Filezilla SCM [6] . In the case of Lighttpd, its Redmine based system [7] holds both the ITS and SCM for the project, giving us access to 2,815 revision changes.

Since our extraction is automated, we included pre-processing to ensure high quality representations of each project, for instance, by removing duplicates. This can be seen as in the Code Change Sets column of Table 5.2 , where the remaining code change sets after pre-processing are shown in the brackets. Although, this pre-processing further reduced the datasets, it also gave us greater confidence in our data by removing false positives.

As shown in Table 5.2, the number of code change sets extracted are relatively low compared to the inspected revision changes. We attribute these discrepancies to missing linkages from the issue tracking system to the revision change log by our automated tool. This is well-known as one of the perils of mining particularly

[1] `http://trac.edgewall.org/`

[2] `http://www.redmine.org/`

[3] `http://trac.WxWidgets.org/`

[4] `http://svn.WxWidgets.org/viewvc/wx/WxWidgets/`

[5] `http://trac.filezilla-project.org/`

[6] `http://svn.filezilla-project.org/filezilla/FileZilla3/`

[7] `http://redmine.lighttpd.net/projects/lighttpd/`

OSS projects, as is also explained by Howison and Crowston [50]. Another reason for missing code changes may be that the revision changes are not related to our dataset versions of code, as the functions do not yet exist or have been modified to a point it is no longer recognizable by our parsing tool.

3.2 Determining Effort Thresholds

To identify high maintenance efforts, we determined effort thresholds by analysing the distribution of the state changes and the duration of the datasets. Since Fig. 5.4 suggests that these distributions do not follow a normal distribution, thus a suitable outlier detection model would be using Tukey's outlier filter [49]. We used the formula $Q3 + 1.5 \times IQR$ to determine the effort thresholds as the outliers for the thresholds $\mathfrak{S}$ (change state) and $\mathfrak{D}$ (duration) as defined at the end of section 2 of this chapter. Q3 is upper quartile of state changes and duration, and IQR being the interquartile range.

Using the grouping thresholds equations (5.1 , 5.2), we are able to separate the issues into a set of high and normal issues. Table 5.3 shows the size of each set, expressed as the cardinality as well as effort thresholds for $\mathfrak{S}$ and $\mathfrak{D}$ for the three projects. As expected, normal maintenance effort is the larger of the two sets. It is interesting to note that there are few high maintenance issues for the Lighttpd project. This could be caused by the relatively longer durations to resolve issues and the lack of state changes of issues as noticed in Fig. 5.4.

Based on the graphs shown in Fig. 5.4 and Table 5.3, the following observations can be made about the *duration* and the *state changes*.

- The density distribution (curve on the graphs) suggested both parameters (state changes and duration) are standard distributions, justifying our methodology for determining effort thresholds.

- As seen in the box-plots for the state change distributions, the state change is constant for all datasets of the same project. This suggests that state change could be constant for each project.

- As shown in the duration distribution for each project, the duration of the maintenance effort varies between projects, and even between datasets

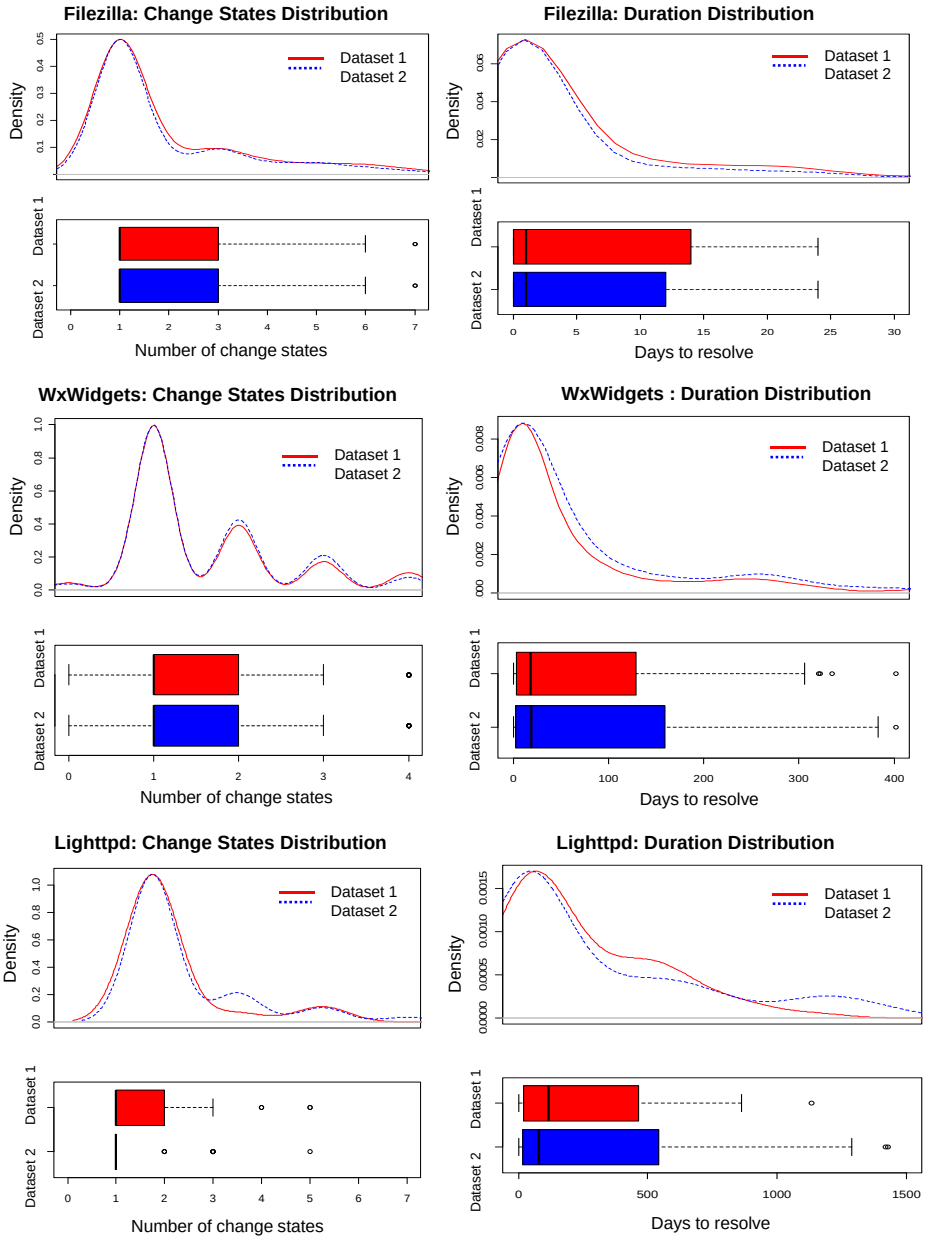
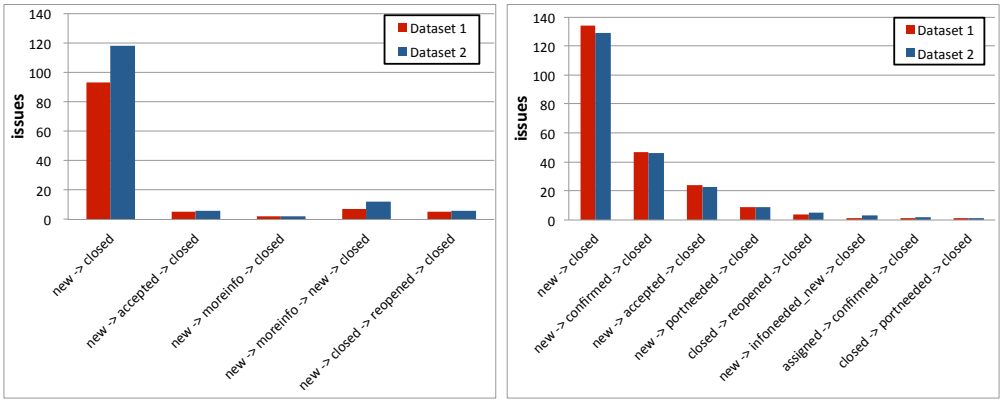


Figure 5.4: Distribution of the datasets



(a) Filezilla (1-3 state changes) (b) WxWidgets (1-2 state changes)

Figure 5.5: Detailed analysis of the state changes for Filezilla and WxWidgets.

within a project. This is also evident for 𝔇 of the WxWidget datasets in Table 5.3

To further understand and validate the state changes, we took a closer look at the state changes lying within the effort thresholds for each project. For both Filezilla and WxWidgets, we had a closer look as shown in the boxplots of Fig. 5.4 where Filezilla has 1-3 state changes and WxWidgets 1-2). To validate that the changing states did represent the workflow we took a closer look at the actual state changes. Fig. 5.5 shows the number of state changes as it represents the workflow for each project. In addition, most state changes are from '*new*' to '*closed*', suggesting a simple *state change*. However, it is interesting that in WxWidgets, there was some relatively moderate use of the '*confirmed*' and '*accepted*' status. Also, in Filezila, the '*moreinfo*' status was relatively commonly used. The analysis confirmed that the issues usually followed the workflow procedures (starting at '*new*' and ending with '*closed*'). It is interesting that some issues in the WxWidget's project start from '*closed*' and '*assigned*' status, causing slight concern in the workflow.

We found that in Lighttpd, the state changes were rarely used with almost 80%-90% of issues extracted with the *New to Fixed* state. There were some instances of other state changes, as shown in Fig. 5.3c. Our results confirmed

Table 5.3: Issues Sets (High/Normal Maintenance Effort) and Effort Thresholds

Project	High Effort (Set Size)	Normal Effort (Set Size)	$\mathfrak{S}$	$\mathfrak{D}$
Filezilla 1	31	69	5	24
Filezilla 2	30	106	5	24
WxWidget 1	96	208	2	190
WxWidget 2	99	204	2	283
Lighttpd 1	5	53	4	1630
Lighttpd 2	10	73	4	1290

that most of the issues do not require a complex process for maintenance effort for the projects, as all three projects mostly use only one state change. However, use of the state changes indicates additional maintenance effort, such as reopening, further information needed or the re-assignment of an issue.

3.3 Metrics Evaluation

Using the effort thresholds specific for each project, each maintenance effort was determined to be either high or normal. Fig. 5.6 shows the matrix of the maintenance effort, expressed in issues, grouped against each metric. Results suggest that high maintenance effort exhibited higher CC and FC, especially using the backward and forward slice metrics.

To prove statistical significance, the student t-test was applied to the groupings. The results shown in Table 5.4, further prove that in all projects the differences in the groupings were significant. Generally the program slicing based metrics have improved p-values compared to the non-program slicing metrics. The program slicing-based metrics were enhanced compared to the *EditedFunction CC* and *EditedFunction FC*. *BackwardSlice CC* and *BackwardSlice FC* outperformed all other metrics.

Based on the results of the t-test as well as the visual representation shown in Fig. 5.6, we can conclude that issues that required high maintenance effort exhibited higher CC and FC values and that the program slicing metrics enhanced this relationship, even in cases where the non-program slicing metrics were not

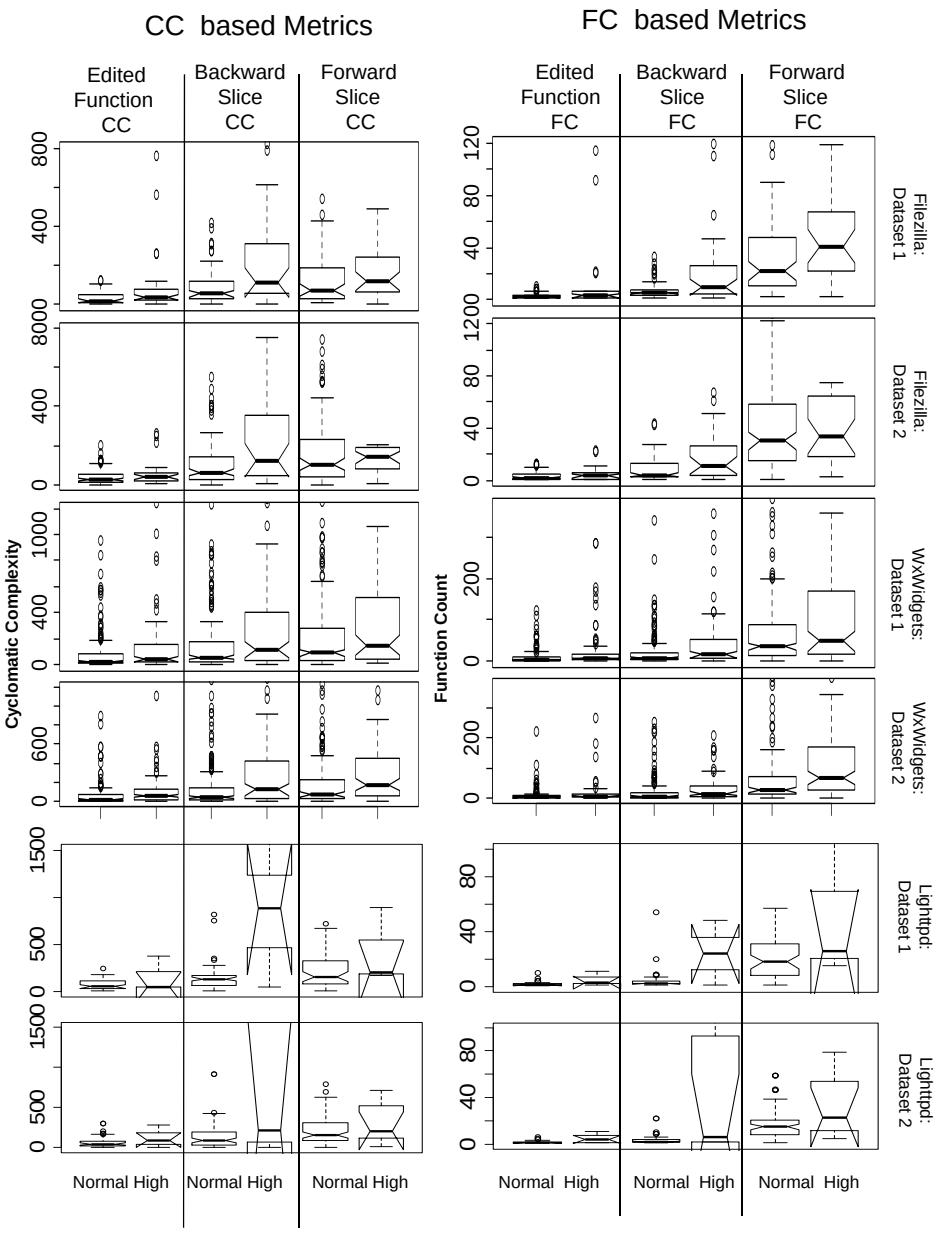


Figure 5.6: Matrix showing the comparison of normal and high maintenance efforts against the metrics.

Table 5.4: P-values for the classic student t-test for statistical significance (p-value less than 0.05). Non-significant values are in bold

Project	Edited Function CC	Backward Slice CC	Forward Slice CC	Edited FC	Backward Slice FC	Forward Slice FC
Filezilla 1	0.001	0.000	0.003	0.001	0.000	0.004
Filezilla 2	0.009	0.000	0.039	0.016	0.000	0.051
WxWidget 1	**0.209**	0.008	0.024	0.005	0.017	0.013
WxWidget 2	**0.111**	0.032	0.019	**0.135**	0.056	0.009
Lighttpd 1	**0.052**	0.000	**0.128**	0.004	0.000	0.006
Lighttpd 2	**0.055**	0.00	**0.196**	0.00	0.000	0.007

significant.

3.4 Other Observations of the Micro Processes

During the pre-processing and cleansing procedures of the datasets, there were a couple of notable differences between all projects. Apart from the obvious workflow and project size differences, under closer examination, we noticed that all the projects had different methodologies in handling issues.

Lighttpd exhibited very high thresholds, especially for duration. For instance, looking at one of the issues, we realised that most fixes are not applied to the source code until the next main version, therefore a normal fix could be resolved, but not committed to code until months later, nearing the next release. More complex issues were being carried over into new revisions before resolved. We also noticed that state changes were being used mostly to identify duplicates and invalid issues rather than track issue states.

For both Filezilla and WxWidgets, an issue is usually submitted with a description of the issue which may also include a screenshot and/or error-log attached. However, it was noticed that almost 90% of WxWidgets issues were accompanied with a potential solution as a patch. Therefore, the WxWidget developer, in addition to verifying that the solution is correct, assesses the impact of the solution on the stability of the current system operation and design. Filezilla has a more traditional approach, with the developer creating the solution based only on information provided by the submitter.

These observations illustrate that even if the projects organisation and manner in which to resolve issues are different our proposed approach was successfully applied.

4. Discussion

In this section we evaluate our approach and how the results could be applied. We then revisit our research questions. Finally we present the threats to the validity of the study.

4.1 Generalizability of our Approach

Our research focused on three OSS projects, all very different in terms of tools, size, workflow and even the micro processes of handing issues as explained in Section 4.3. Yet, our approach and metrics yielded consistent results, suggesting that our approach is feasible across OSS projects. Our project-specific effort threshold method enables us to expand our approach to various projects. Also, our study proved that this approach can be applied to projects ranging up to 400 kLOC.

From the study, we identified two factors that could influence the effectiveness of our approach. The first one is the availability of the data. Smaller projects or newer projects may not have sufficient data for analysis. Also, we assume that more mature projects would have more data for analysis. Furthermore, although we have determined that this approach can be applied to projects up to 400 kLOC, this study did not determine the lower limit needed for reliable results. The second factor is the linkages between the issue management system and its source code management system. Detailed documentation and linkages from the SCM and ITS is essential for our approach. It is envisioned that as technologies improve the documentation and their tracking of linkages between SCM and ITS, our approach will become more feasible.

4.2 Slicing Evaluation

The results in Table 5.4 suggests that the backward slicing outperforms forward slicing. Firstly, we believe the relationship between the size of backward slice and the maintenance effort taken to find and fix the bug is directly related. However, when considering forward slicing, there is a risk of the fix incurring side effect bugs. We believe this risk could be the cause of the inconsistent results. We did not validate this, however, this could be one possible reason why the backward slice metrics is a more reliable metric.

4.3 Applications of the Study

As an empirical study, our work has shown that the proposed methodology is able to determine the micro processes requiring high maintenance effort, based on the combination of duration and complexity of the micro processes. There are two major viewpoints where we can utilize these results, from both a source code maintainability perspective as well as the project process management point of view.

Code-based Application.

Identification of high maintenance efforts as well as the affected code, especially those with relatively high CC and FC characteristics, could be beneficial from a source code maintenance point of view. These code portions could be candidates for source code maintenance activities such as refactoring, code reviews and code inspections. These activities would improve the quality of the source code, thus ensuring that the complexity of code is kept low. Since there is a correlation between complexity and maintenance effort, these activities have potential to reduce the maintenance efforts. However, it is important to note that we only suggest a causative correlation. Investigation of a causative link is viewed as future work. In addition, the affected code could be marked as complex, so that a developer can take extra care when modification of these portions of code are required.

Project Management-based Application.

From a project team standpoint, the ideal scenario is to assign the best developers and resources to the code changes that require the most effort. To do this, a team first identifies high maintenance efforts at any point and on a particular version and compares this to maintenance efforts of previous versions. The team then assesses if their current workflow is still suitable for the project. Through this, the team can be reorganized so that the most experienced developers and resources are assigned to *'high maintenance effort'* portions of code.

More specifically, in projects similar to WxWidget's method of handling issues (i.e., in which a potential solution accompanies the issue), the developer can already identify which functions will be edited and consequently assess what level of effort (more experienced developer) should be assigned to this issue beforehand. In regard to projects like Filezilla and Lighttpd, there could be an additional *verification* micro process, in which functions that have potentially high CC and FC can be validated by the more experienced members of the project team.

Our approach identifies issues that require high maintenance effort also have considerable change impact on source code. However high effort due to longer durations to resolve issues can be influenced by other factors, including back-burners (referring to issues that are of low priority because they do not greatly affect the system) or simply poor documentation creating a delay in assignment or breakdowns in the micro processes. Still, this approach can inform project managers of the current quality of the implemented software processes, giving useful insights to improve the micro processes. Also, analysis of the changes in the effort thresholds over versions, although not in the scope of this study, could prove useful.

4.4 Research Questions revisited

Before this study we constructed three research questions in relation to our assumption that high maintenance effort is related to complex source code. In this section we revisit each question against the results of the study.

- ***RQ1.*** *Are we able to determine high maintenance effort?* By analysing micro processes during the maintenance phase, we were able to use an effort

threshold to differentiate and group issues that consumed *high maintenance effort*. Our approach proved that this threshold is project-specific.

- ***RQ2.*** *How much impact does high maintenance effort have on the source code?* Results of the study indicate a correlation between high maintenance effort and code changes with high complexity metrics (proposed FC and CC based metrics).

- ***RQ3.*** *Is there a correlation between maintenance effort and its impact to source code?* Applying the standard student t-test, results proved a significant statistical correlation between maintenance effort and the impact of its changes to the source code. In some cases, our proposed metrics outperformed the non-program slicing metrics.

Relative to one of Lehman's laws of evolution [66], we can logically conclude that higher maintenance effort would most likely be caused by complex code. This may seem trivial and common sense, however, in this study, our novel contribution is that we have quantitatively express this established phenomenon.

Threats to Validity

Internal

The major internal threats would be the accuracy of the tool used for extraction and the measurement of the data, since the extraction process is automated. To mitigate this, a combination of manual verification and pre-processed data was used to ensure the quality of the data collected. It was noticed that the amount of data extracted is significantly lower than the expected amount of code changes available. This can be attributed to the missing linkages between ITS and SCM as well as the lack of proper documentation.

Another threat could be that the pre-processing of the data has an influence on the results. In a real world scenario, for example, data is usually not pre-processed. We believe, however, that our pre-processing was based on rules that can be applied automatically in the real world, as they are not based on human judgement.

Research by Binkley et al. [21], indicated that as data structures increases, so does the size of the slice, causing accuracy problems. In this study, we did not address this issue as most large slices (having large FC and/or CC) were identified as having valid complex interprocedural dependencies. Also, we believe our case study datasets are under 1 million LoC, therefore not considered large scale projects. However, we see this as an important issue, especially when looking at large scale programs and regard this as future work.

External

The major external threat to validity is the generalization of our approach to different projects, under different software repository systems, and since the micro processes and resolution workflows differ from project to project. Our effort threshold method offers project specific measurements to take into account these differences. In addition, since these are only three projects, we are uncertain if this is a true representation of an OSS project. Our data is all from OSS projects, hence we cannot assume that results may be the same for commercial projects. Generalization of our techniques is seen as the most important priority for future work.

In this research, we focused solely on change impacts in source code, as we believe maintenance of software mainly revolves around source code changes. Our definition of maintenance effort also investigated micro-processes during maintenance. However other aspects such as software architecture issues were not taken into account. This could be interesting avenues for future work.

Another threat is that our tool used to extract data was specifically made for the TRAC and Redmine (ITS) as well as SVN SCM systems. We would like to expand to other tracking systems such as bugzilla ITS and CVS SCM systems. We plan to further develop our tool to handle other systems so that we can apply our approach to a wider set of projects.

As mentioned previously, there are not many available projects with sufficient linking information of the software repositories to date. We, however, believe that the tools and technologies to manage documentation of software repositories are steadily improving. This makes our approach more practical in the near future.

5. Chapter Summary and Future Work

Software process assessment and improvement can be a rather complicated and costly exercise that is only suited for larger organisations. Measuring software quality is also complicated and covers a wide range. In this study, we present a simpler approach: focus on the micro processes of maintainability of the code.

To the best of our knowledge, this work is the first to express maintenance effort in terms of the complexity of the micro process. By using effort thresholds, we were able to determine which micro processes required high maintenance effort. Also, using novel program slicing-based metrics, we were able to measure the impact of these efforts on the source code. We concluded that there is a statistically significant relationship between maintenance effort and its change impact on the source code.

Although the results are promising, there are still outstanding issues for future work, including the following:

- *Generalization of our approach.* Replication of the study with more OSS projects is needed.

- *Explore the correlation between maintenance effort and impact on source code further.* Currently we have identified a correlation but whether this is causative is another avenue for research

- *Explore the change impact to handle large-scale systems.* Our program slicing based metrics outperformed the non-program slicing metrics, however we need to address the issue what to do when program slices become too large. We plan to study strategies to break down these structures, so that program slicing is more manageable.

- *Study the effort threshold behaviour over a project's lifetime.* The change in effort thresholds over different releases of a project could potentially measure the state of the maintenance effort during the duration of the project.

- *Assessment model for the assessment of the maintainability of software.* Although we are only in the early stages of validating the generalizability

of our approach, the final goal would be to create an assessment framework and prediction model for this assessment.

In this study we explore the relationship between process and product. An interesting caveat is that as model-driven software development approaches evolve, and the automation in software evolves along, model elements instead of the source code will be automated. In such a scenario, the relationship between the process and product will be an promising research avenue.

As data management processes, tools and techniques improve, we envision this line of research to prosper. We see this study as a step towards a viable quantitative approach for software process assessment of the maintainability of software.

Chapter 6

Case Study: A Controlled Experiment

1. Introduction

In the previous chapter, we demonstrated the application of our approach to three OSS projects. Most open source projects record maintenance effort coarsely as the number of days, which we found can be affected by many outside factors such as priority, developer's workload and the maintenance work-flow (process) complexity. In this chapter we eliminate these factors using a controlled environment setting.

According to one of Lehman's laws of evolution [66], we assume that higher maintenance effort is most likely caused by complex code. We believe that the analysis of source code properties and how they correlate to the maintenance effort during the resolution of an issue could show this quantitative. To test this theory, we aim to answer the following research questions:

- ***RQ1.*** *Is there a quantitative relationship between maintenance effort and source code properties?*

  Assuming that there is an evident relationship from RQ1, we then constructed the next research question:

- ***RQ2.*** *Can we define a set of code-based metrics from this relationship?*

Applying our proposed metrics introduced in Chapter 4, we analyse the relationship.

Our final research question is concerned with the performance our program slicing based metrics against the conventional (non-program sliced) Lines of Code (LoC) and CC (Cyclomatic Complexity).

- ***RQ3.*** *How do the metrics compare in degree of relationship?*

To evaluate our research questions and approach, we performed trial experiments on a set of pre-defined maintenance issues. We designed four issues, each representing the specific type of maintenance activity commonly encountered in the real world. We measured maintenance effort based on the duration to resolve each issue, without explicit maintenance processes such as workloads, work-flow and other environmental factors.

The preliminary results from our trial experiments suggest that the proposed program slicing metrics have the strongest correlation with maintenance effort, exhibiting a moderate to strong degree of correlation with maintenance effort against our proposed metrics. In contrast, the conventional LoC metric had a very weak correlation with maintenance effort.

We envision that our approach will give quantitative insights on where maintenance efforts can be reduced. This could be done by the proper assignment of resources and the identification of high maintenance-prone portions of code as candidates for maintenance activities such as refactoring, code inspections and reviews.

2. Approach

In this study, we decided to use a controlled environment to cover the different types of issues. There are several advantages: (1) Using a set of pre-determined issues allows to expose the different types of typical code changes. (2) In addition, we can pre-define the complexity of the code change. (3) We can measure accurately the duration taken to resolve the issue. Normally, the duration to resolve an issue can be influenced by several factors such as the bug fixing process

Table 6.1: AlignMe function descriptions

Function	Description
main()	runs the program, displays menu to console and calls up the Align instance
Align::doIt() Align::setStrategy(int)	calls up Strategy::format() function based in input creates either an instance of RightStrategy or LeftStrategy
Strategy::Strategy() Strategy::format() Strategy::justify(char*)	constructor for strategy class reads the text from quote.txt and calls justify(char*) for either RightStrategy or LeftStrategy virtual function is implemented by either RightStrategy or LeftStrategy
RightStrategy::RightStrategy() RightStrategy::justify(char*)	constructor implements the specific right alignments for the text
LeftStrategy::LeftStrategy() LeftStrategy::justify(char*)	constructor implements the specific left alignments for the text

or work-flow, workload of the developer and in some instances the priority of the issue.

This study specifically ignores all these factors, so that the maintenance effort is solely based on the time given to resolve. Thus, $\mathfrak{D}$ is the time to fix (minutes), thus maintenance effort $\mathbb{ME}$ is defined as:

$$\mathbb{ME} = \mathfrak{D} \tag{6.1}$$

The layout of our approach is as follows. First, we introduce the proposed metrics used to measure the source code properties of the code changes. Later in the section, we introduce the experiment with the four issues that were used in the experiment and the feedback questionnaire.

To generate our metrics, the software analysis tool by GrammaTech called CodeSurfer [5], arguably the most sophisticated and widely used tool for inter-procedural slicing, was used [88, 4, 22]. Customized scripts within CodeSurfer were used to calculate the CC and FC within the slices.

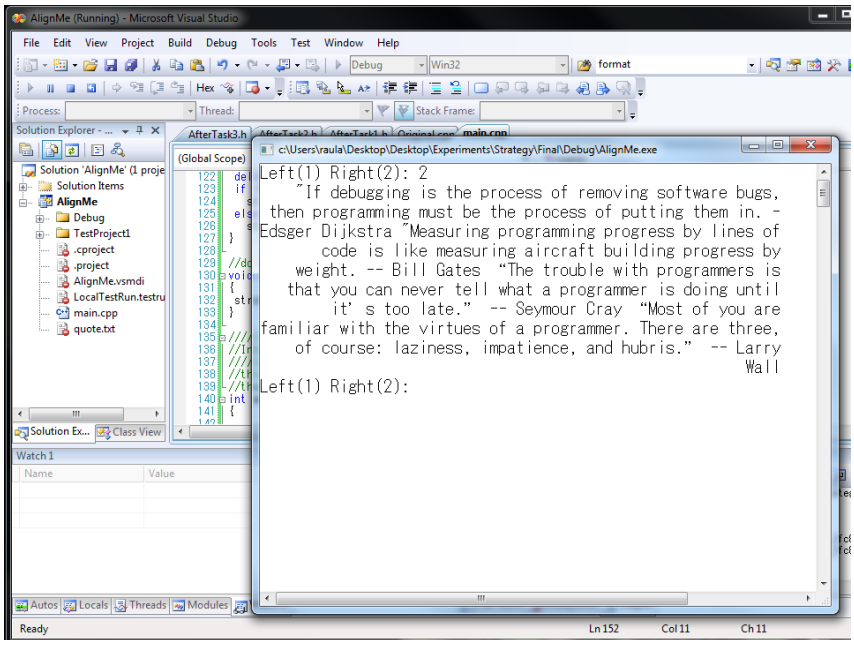


Figure 6.1: Screenshot of AlignMe showing the right alignment

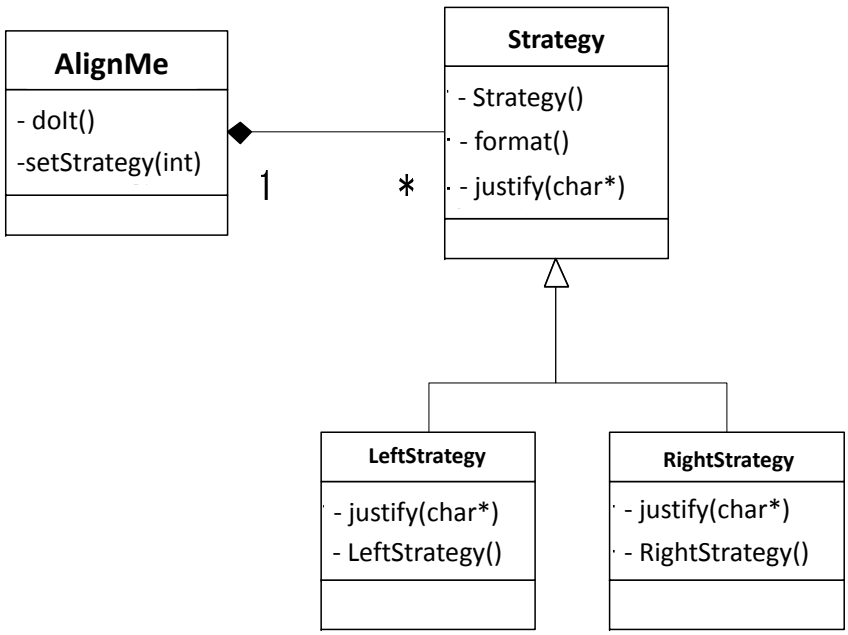


Figure 6.2: AlignMe class diagram. Note that *main()* is used to instantiate the align class

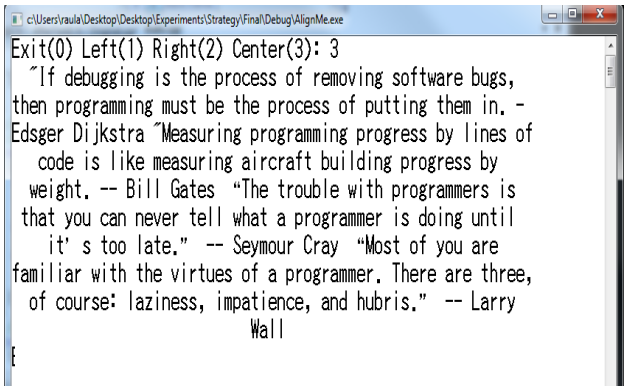


Figure 6.3: Screenshot of ideal solution for issue 1. This should be the output with the center option is selected

3. Pilot Experiment - AlignMe

Software maintenance and evolution are inevitable, with constant changes to code. According to Yu [118], there are three main software maintenance activities: (a) Perfective - the addition of a new functionality, (b) Corrective - fixing of faults or bugs to the software and (c) Adaptive - new file formats or refactoring code. In this study, we only focused on the first two activities due to two reasons. Firstly, due to time limitations of the experiment. Secondly, because of the difficulty in designing adaptive maintenance issue. We designed four issues: Issues 1 and 2 were perfective activities, while issues 3 and 4 addressed some corrective activities.

Another major challenge encountered involved defining a set of issues were that they had to be solved within a time-frame by novice users to both the program and the programming language. Taking these factors into account, we developed the testing program AlignMe. For easier code understanding and appending new enhancements to the program, we incorporated software design patterns into our program.

AlignMe is a console-based application written in C++. It was developed and run in Visual Studio 2008 Environment on the Windows 7 platform. The main function of AlignMe is to display text from a stored file (quote.txt) in either a selectable Left or Right alignment. AlignMe has a total of 78 LoC with 4

classes consisting of 10 functions. Table 6.1 shows a summary of the functions and their descriptions. The class designs in Fig. 6.2, illustrate how the strategy design pattern is implemented. The strategy design pattern is commonly used to improve the readability and maintainability of source code. As seen in Fig. 6.2, the class Strategy is the parent of classes RightStrategy and LeftStrategy.

The following are the maintenance activity issues designed for the experiment:

- **Issue 1 - Create a center alignment** At the current state, program offers the two options of either right or left alignment. Participants are asked to modify the program to add a center alignment. Fig. 6.3 illustrates the expected output. This is a perfective activity, which the ideal solution involves extension of the strategy design template to include a CenterStrategy class from the parent Strategy class, with appropriate changes in the justify() implementation. This issue was designed to have the largest size in terms of LoC.

- **Issue 2 - Allow user input of layout width** At this state, the program has a fixed width for the layout of the text. Issue 2 enables the user to customize the layout width. This is a perfective activity that is designed to generate high FC and CC metrics by including modifications to many functions. Fig. 6.4 shows the desired output. According to our proposed metrics, this issue should take high maintenance effort as having the highest FC and CC metrics.

- **Issue 3 - Allow subsequent alignment input** At this state, the program only allows a single alignment before exiting automatically. Issue 3 requires participants to allow the user to run several alignments in one session before choosing to exit. The issue is a bug because even though returning to the menu after alignment, it immediately exits if any key is pressed. Fig. 6.5 shows an example of the desired output, where the alignment is run three times. The ideal solution would be adding a for-loop to the main function, so the program continues to return to the main menu until the exit option is selected. This issue was designed to be a simple fix that only affects the *main()* function.

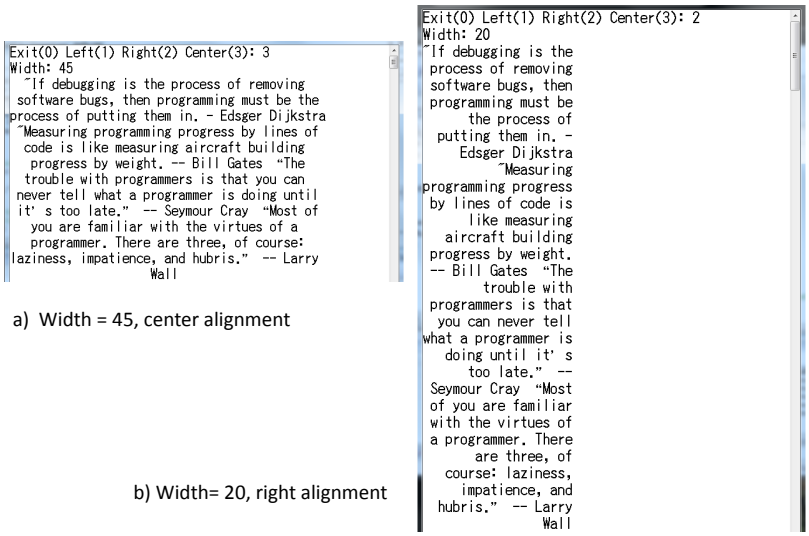


Figure 6.4: Screenshot of implementations of the ideal solution for issue 2. a) shows a width of 45 while b) has a width of 20.

- **Issue 4 - Buffer overrun error** Since issue 2 allows user input of the layout, this creates another bug since the program cannot handle layout widths greater than 60. Issue 4 requires modifications to allow widths up to 80. In this case, the solution requires the user to manipulate the array size of the variables storing the text. This has to be done in classes RightStrategy, LeftStrategy and CenterStrategy. It is a simple fix, however, it requires the developer's comprehension of the program.

4. Analysis and Evaluation

Using the proposed metrics, we then performed a quantitative analysis of the issues previously mentioned. Using our proposed metrics we evaluated each of the maintenance activities.

Taking a closer look at LoC from Fig. 6.6, we can see that issue 1 has the most modified lines, while issue 3 has the lowest. In contrast, according to the program

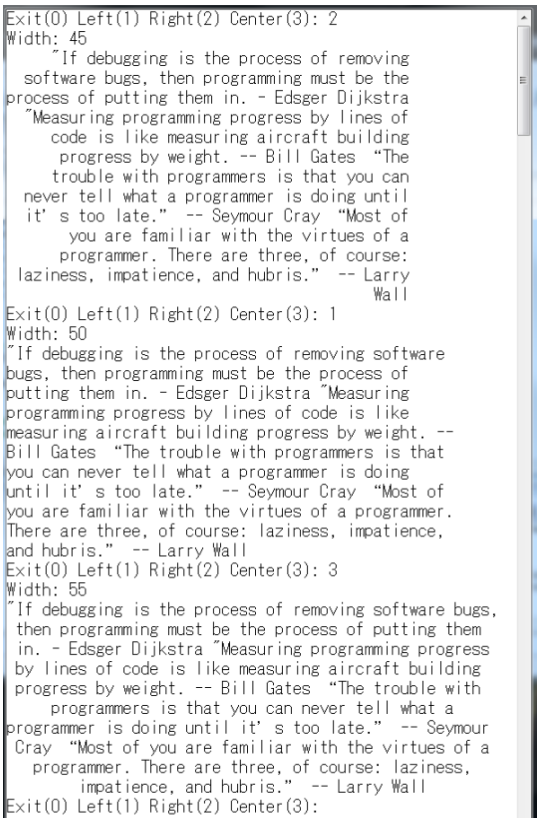


Figure 6.5: Screenshot of ideal solution for issue 3. The program is able to execute three times before exiting.

slicing metrics shown in Fig. 6.7, issue 2 has the highest CC and FC values for both slicing and non-program slicing metrics. The metrics are contradicting, because while the CC and FC metrics suggest issue 2 as candidates for high maintenance effort, LoC metrics, on the other hand suggests issue 1.

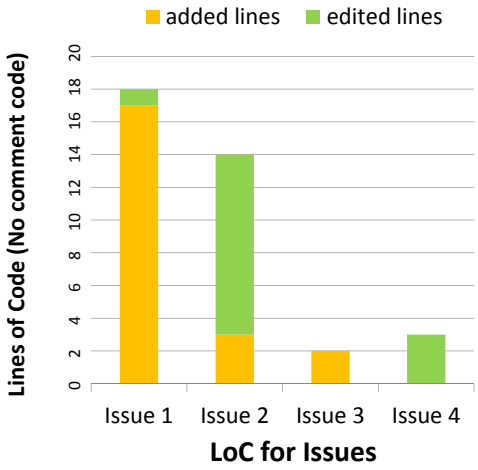


Figure 6.6: This figure shows the quantitative analysis of the issues related to LoC.

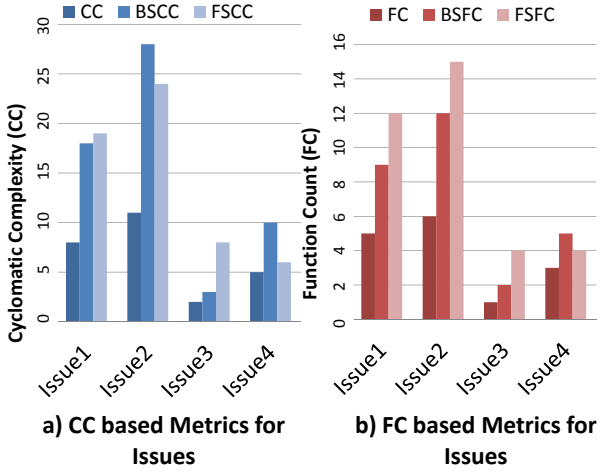


Figure 6.7: This figure shows the quantitative analysis of the issues for our proposed metrics. Note that (a) CC based metrics and (b) FC based metrics.

5. Results

5.1 Participants

A total of eight test subjects participated in the experiment. All participants had a strong programming background and were currently graduate students from the

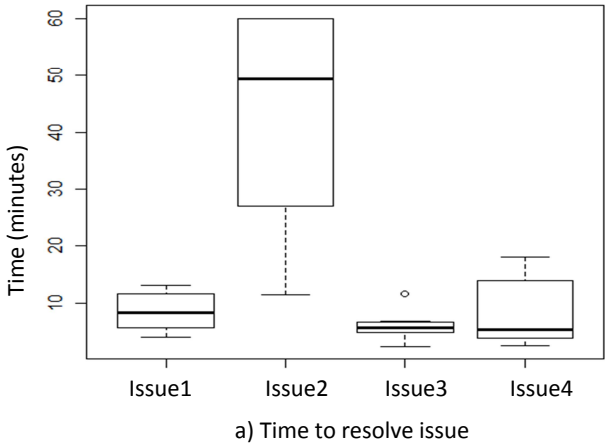


Figure 6.8: This boxplot shows the distribution of time taken to resolve each issue during the experiment.

Information Science department. In fact, three of the participants had previously been programming instructors, while the rest had worked on small to medium software projects. Four of the test subjects preferred C++ while the other four listed java as the preferred programming language.

5.2 Experiment Environment

During the experiment, each participant was given a 10 minute pre-experiment tutorial, explaining the technical aspects of the program. After this, an additional 5 minutes was then allocated for questions and/or free time to get familiar with the program before the maintenance activities. Each task was introduced in the same sequence for all participants along with screenshots of the desired output as seen in Fig. 6.3, 6.4 and 6.5. Each task is marked as resolved only after testing for the desired output. Only then, the next task was introduced. After all the tasks were completed, a simple questionnaire was filled out by all participants to collect comments, preferred programming language and difficulty of each issue. For all participants, the same isolated experiment environment was used.

Please refer to Appendix A for a sample of the tutorial and default answers of the experiment.

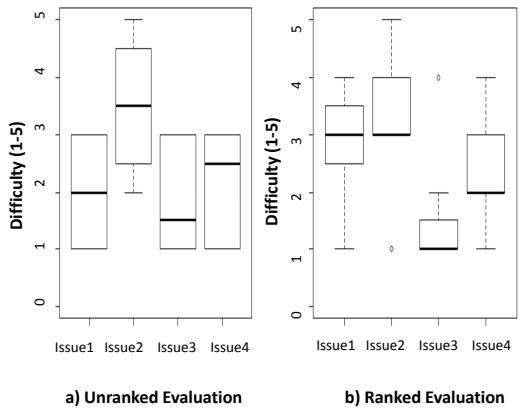


Figure 6.9: (a) shows the perceived difficulty of each task by participants and (b) is the ranking order of difficulty by participants. All rankings are from 1 =easy to 5=hardest.

Table 6.2: Spearman's rank correlation of proposed metrics with maintenance effort

	LoC	**CC**	**BSCC**	**FSCC**	**FC**	**BSFC**	**FSFC**
Other (non C++) Developers	0.27	0.58	0.61	0.50	0.51	0.57	0.51
C++ Developers	0.45	0.78	0.83	0.79	0.69	0.78	0.78
Total Developers	0.35	0.66	0.70	0.63	0.58	0.65	0.62

5.3 Evaluation

For each participant, the average time for completion of the whole experiment was within 2 hours, with each issue ranging from 10-60 minutes for completion. Fig. 6.8 shows the distribution of the time taken to solve each issue. Fig. 6.9 (a) shows the individual ranking of each issue with a score from 1-5, while Fig. 6.9 (b) shows when they are ranked in the order of difficulty. These graphs clearly suggest that issue 2 was the hardest issue to implement. In reference to issue 2, one subject commented that the " *C++ syntax used to pass arguments across functions was the problem*". Other responses noted that unfamiliarity with the programming language lead to difficulties with issue 2.

In regard to the rest of the issues, participants identified them as typical maintenance activities. We assume that the difficulty of issue 1 was greatly aided by the use of the strategy design pattern as pointed out by a participant. Also, it was

interesting to note over 75% of the participants had limited knowledge on design patterns, but found it very useful. Issues 3 and 4 were noted to be typical code changes. Issue 3 was seen as the easiest as it only involved minor modifications within a single function. Issue 4 took slightly more time as modifications had to be made in several locations, but not significant.

6. Discussion

Our research investigates a quantitative approach to measure maintenance effort using a set of proposed metrics. In our previous work, we studied real world projects, with results suggesting false positives of maintenance effort, which take more time to be resolved due to the following factors: (1) low priority (back-burner), (2) human resource assignments or even a (3) delay feedback in the bug fixing work-flow

Analysis of the issues in Fig. 6.6, 6.7, 6.8 and 6.9 suggest that issue 1 is most likely candidate for high maintenance. However, our proposed metrics instead suggested issue 2 and was proven by our experiment results.

A controlled environment was used in which other factors such as the bug fixing process, developer workload, and priority were nullified. Table 6.2 proves our proposed metrics outperform the conventional metrics. This is significant as in real life the correlation may not be as strong. The correlation could serve as guidelines to quantitatively assess maintenance effort and processes, outperforming traditional measures such as LoC. The usage of these metrics can be applied from two viewpoints.

From a code-based perspective, identification of potentially high maintenance effort code portions could be candidates for source code maintenance activities such as refactoring, code reviews and code inspections. These activities would improve source code quality and maintainability. Correlation between complexity and maintenance effort, these activities have potential to reduce the maintenance efforts. In addition, the affected code could be marked as complex, special attention is taken during code modifications.

From a project team standpoint, proper assignment of the most proficient resources to be made to these complex code changes. Teams can manage and

assess their current work-flow or maintenance processes.

6.1 Revisiting Research Questions

We address our research questions below:

- ***RQ1.*** *Is there a quantitative relationship between maintenance effort and source code properties?* To evaluate our results, we used the spearman's rank correlation as the data is non-parametric. Since familiarity with C++ was perceived to be an influential factor, we classified subjects for participants that were comfortable with C++ (C++ developer) and those not (other developers). Table 6.2 suggests moderate correlations for each set of metrics.

- ***RQ2.*** *Can we define a set of code-based metrics from this relationship?* As mentioned in RQ1, our experimental results suggest our proposed metrics have a moderate to strong degree of correlation with maintenance effort.

- ***RQ3.*** *How do the metrics compare in degree of relationship?* As seen from Table.6.2, we see a positive association of maintenance effort with the proposed metrics (except of LoC) in all groupings. It is shown that the program slicing-based metrics have the strongest correlations, with backward slice-based metrics (BSCC - up to 0.83 and BSFC - up to 0.83) had the strongest degree of correlation. We found that all groupings have similar differences of correlation for all metrics, thus strongly suggesting a trend.

6.2 Threats to Validity

The main threat is whether our experiment is a true representation of the real world. A program comprising of 78 lines of code is not a representative of real world programs, however, we believe that we fundamentally represented basic maintenance tasks that could be sufficient for a typical programmer could attempt without prior knowledge of the program. Also in this study, the program slices were manageable, which may not be the case in the real world. Program slicing is complex, especially with large systems and this issue is being considered for

future work. Another threat was the assessment of participant's experience and skill. Keeping this in mind, the issues were designed so that they could be solved within a reasonable time-frame. We designed the program so that it would be easily understood by typical programmers.

Internal threats were the accuracy of the data collected. The consistent trend of the performance of the metrics across the three groupings of correlations gives us confidence in our results. Another internal threat would be our program slicing tool. However, as mentioned earlier in the paper, CodeSurfer is one of the most widely used tools for program slicing.

7. Chapter Summary and Future Work

The purpose of this research is to investigate an alternative quantitative approach of the analysis of maintenance effort. In this study, we quantitatively expressed this established phenomena, using several common code-based metrics as well as our proposed program slicing-based metrics. Although the results are promising, there are still outstanding issues for future work, including the following:

- *More application to the real world.* For future work, we would like to apply our approach to industrial projects.
- *Explore other factors of maintenance effort.* Evaluate the impact of factors such as the bug fixing process, developer workload, and issue priority. Also, explore differences between the non-slicing and slicing based metrics.
- *Explore the change impact to handle large-scale systems.* Further investigate strategies to manage large slicing. Heap slicing is a possibility.
- *Assessment model for the assessment of the maintainability of software.* Other aspects such as process effort, human and infrastructure management for the achievement of process objectives will be investigated.

Although, the study is in its early stages of validating the generalization of our approach, the final goal would be to create a process assessment framework and prediction model.

Part III

Peer Review Processes

Chapter 7

OSS Peer Review Process: Models and Metrics

1. Introduction

In this chapter, we proposed metrics and models to assist peer review members identify their current standing and career advancement opportunities in an OSS project. Using MPA (Micro Process Analysis), we proposed expert profiling and career trajectory metrics and models. This work contributes to the adoption of SPI (Software Process Improvement) models that are driven at the developer level.

2. Theory and Related Work

A number of prior studies have shown the effectiveness of code reviews. For instance, Boehm and Basili [25] noted that review typically catch 60 percent of product defects. Also, Mantyla and Lassenius [69] also discovered that 75 percent of defects found during peer reviews improve the software evolvability by making it easier to understand and modify.

There is a large body of research on peer reviews in OSS [27], [100], [104]. According to Aberdour [1], there are differences in OSS quality assurance as compared to closed source projects. With code reviews, instead of the traditional

round table meetings with physical developers, potential code changes are submitted as *patches* via email or review systems for review by core experienced members of the community. OSS code review processes have been gaining popularity as it ensures quality, especially in regards to contributions from outside the core community [95]. In fact, Aberdour states that OSS projects seem to "eschew best practices without software quality suffering". Futhermore, OSS reviews could be superior due to the larger pool of reviewers [11]. This is since Linus law [94] that implies that " *given enough eyeballs, all bugs are shallow,* " meaning that if enough people see a software error, at least one of them will probably understand the cause and be able to fix. Also, Gacek and Lawrie observed that OSS development faces fewer time and cost pressures than closed-source development [65].

However, the main difference between OSS projects to the closed software projects is that contribution is of a voluntary nature. High-quality OSS relies heavily on having a large, sustainable community to develop code rapidly, debug code effectively, and build new features. Many studies concluded that creating a sustainable community should be an OSS project ' s key objective [94], [104]. Several papers such as Lakhani [64] state that suggested motivations include the personal need for the software, reputation-seeking, and altruism. This is unlike traditional software projects, where knowledge and the presence of mentors are crucial [102], OSS depends solely on developer's self interest and motivation. Mockus et al. [76] studied Mozilla-a highly modular project that initially had trouble attracting contributors. They found that participation increased only after the core team improved documentation, wrote tutorials, and refined development tools and processes. This is an example of the importance of attracting and sustaining contributor's motivation in an OSS project. Additionally, due to the distributed nature of OSS, member's lack the conventional face-to-face interaction and structure compared to closed projects. This lack of visibility could prove detrimental towards members motivation and sense of position within the community.

Much OSS literature has been performed on different aspects of the OSS peer review. Researchers have focused on the OSS review process and organisational structure [11], [95], and its social technical aspects [95], [24], [32]. We assume

that most studies have not attempted to profile members.

Finding and profiling experts is not a new concept and used in knowledge management and collaboration [14] in other fields such as artificial intelligence. For example, MITRE created an expert finder [1] to help users quickly locate people who know about a particular subject area. Other implementations are with the peer review process for journals and conferences. The expertise recommender [72] and expertise browser [76] are examples of approaches to identify implementation experts. Similar to these approaches, we used data mined from the software repositories to determine our member types.

Career path analysis and trajectories have been used in other fields such as biology and implemented in the social sciences and economics [97], [70]. Two well known state-of-the-art approaches are hierarchical clustering and categorical based on factorial analysis. Sequence mapping techniques such as self-organising maps (artificial neural networks) and Markov chains have been proposed to map pathways of state transitions [97]. We assume that this has not been yet applied in the OSS peer review setting.

3. The Android Project

The Android Open Source Project (AOSP) is a linux-based operating system for mobile devices such as smartphones and tablet computers, developed by Google in conjunction with the Open Handset Alliance [2] . The first Android-powered smartphones were sold in Q1 2009, and has since grown to become the biggest smartphone operating system.

AOSP currently has a public and private branch for members to contribute patches. In this study, we solely focus on the public branch. Using custom scripts, peer review data was extracted from the online android gerrit code review system [3] .

AOSP uses the GIT source code management system in conjunction with gerrit, a web-based collaborative code review tool. Gerrit automatically records

[1] http://www.mitre.org/news/the_ edge/june_98/third.html

[2] http://source.android.com/

[3] https://android-review.googlesource.com/

and tracks all merges into the source code, including details related to the code patch and the peer review process activities. To track all peer review activities, we extracted all patches regardless of current state (abandoned, merged or still open). For each patch report, we used specific features for our analysis. For identification, we extracted *the patch id* as well as the *member name, id* and the *email address* of the patch owner. We then extracted all the contributors involved in that patch report, identified as *submitters, reviewers or verifiers.*

3.1 Terminology

From the AOSP documentation [4] , we present the following terms to be used throughout the paper:

- **Member.** A `member` of an open source project is an individual that performs any of the peer review activities.
- **Peer Review Activities.** These activities are usually the roles of: i). contributor, ii). Reviewers and iii). Verifiers/Approvers. We consider the submitting patches, reviewing patches and verifying/approving patches as peer review activities.
- **Contributors.** A `contributor` is anyone that makes contributions to source code, thus submitters of patches. In this paper, we assume that the contributor is the owner of the code, which in reality is not always the case.
- **Reviewers.** `Code reviewers` are contributors that can review submitted patches.
- **Verifiers and Approvers.** A `verifier` is responsible for testing patches. Contributors can be invited to become verifiers after contributions of significant high-quality code. `Approvers` are experienced members who are said to have made significant contributions and were previously verifiers. This role is by invitation from project leads and has the power to include

[4] http://source.android.com/source/roles.html

of exclude changes, verifiers as assumed to have a higher social status over members.

- **Project Leads.** `Project leads` are responsible for leading all technical aspects of the project and assign verifiers and approvers. Typically they are google employees.

- **Time-frame.** In order to track member's review activities over a period of time, we need a time-frame to measure activity. During this time-frame there are intervals in-which the activity is assessed and measured.

4. Peer Review Profiling Metrics

According to Mockus [77], expertise is strongly related with experience. Building on this, we based all of our metrics on membership duration. Also, Bird [24] states that member's review activities follow a hazard rate, dropping activity levels after attainment of high social status. To incorporate this phenomena, our proposed metrics are designed to provide a contributor's activity performance in relation to their tenureship. We measure the activities as a daily rate, thus identifying inactively of members over time.

Each metric is defined using the following attributes:

- **Tenureship** - This is a measure of duration (per day) since a member had joined the project, from the first day till the latest update. Three different review activities are considered: i). review of a patch, ii). submitting a patch, or iii). verification of a patch. Members with outstanding tenureship are referred to as *Seniors (T)*.

- **Submit rate** - This is a measure of how many patches the member has submitted. We called members with outstanding submissions *Submitter(S)*.

  The derived metric is defined as:

$$\frac{numberofSubmissions}{Tenureship} \tag{7.1}$$

Table 7.1: Expert Matrix: T= Tenureship, S=Submits, R=Reviews, V=Verifications, X= Extreme Attribute

Expert	Member Types	T	S	R	V
Verifier	**Core Member (TSVR)**	X	X	X	X
Verifier	**SeniorVerifyingSubmitter(TVS)**	X	X		X
Verifier	**SeniorVerifyingReviewer(TVR)**	X		X	X
Verifier	**ActiveMember(SVR)**		X	X	X
Non-verifier	SeniorSubmittingReviewer(TSR)	X	X	X	
Verifier	**SeniorVerifier(TV)**	X			X
Verifier	**VerifyingSubmitter(VS)**		X		X
Verifier	**VerifyingReviewer(VR)**			X	X
Non-verifier	SeniorSubmitter(TS)	X	X		
Non-verifier	SubmittingReviewer(SR)		X	X	
Non-verifier	SeniorReviewer(TR)	X		X	
Verifier	**Verifier(V)**				X
Non-verifier	Reviewer(R)			X	
Non-verifier	Senior(T)	X			
Non-verifier	Submitter(S)		X		
Non-verifier	Non-expert(N)				

- **Review rate** - This is a measure of how many patches the member has been involved as a reviewer. We call members with outstanding reviews *Reviewer(R)*.

  The derived metric is defined as:

$$\frac{numberofReviews}{Tenureship} \tag{7.2}$$

- **Verify rate** - This is a measure of how many patches a contributor has verified and approved before merging into the source code. We refer to these members as *Verifier(V)*. We assume that experts should be at least verifiers, as they are able to approve new code. The derived metric is defined as:

$$\frac{numberofVerifications}{Tenureship} \tag{7.3}$$

Table 7.1 shows a matrix of all the possible combinations of the proposed metrics, from which we derived different *member types*. From our metrics, we

identified a combination of 16 different member types, each categorised by the number of extreme attributes identified.

Table 7.1 also shows that 8 out of the 15 classifications are verifiers. As mentioned in the previous section, verifiers have higher authority, thus could be referred to as being the experts.

4.1 Threshold Attributes

In order to identify members with specific outstanding activity attributes, we defined *threshold measures* using the `pareto principle`, as our baseline measure. The pareto rule, a concept from economics with applications in engineering, states that roughly 80% of the effects come from 20% of the causes.

Activity rates can be misleading for new members, due to their low tenureship. We defined a minimal threshold (`lower limit`) so that a member is only eligible for categorisation after a defined tenureship duration. We applied the same 20-80% ratio. Thus, members are required to have tenureship greater than 20% of the population before being eligible.

5. Peer Review Empirical Models

According to the proposed metrics and the threshold attributes mentioned in the previous section, we were able to profile and classify into member types.

In this paper, we study member's historic transition of member types to form the career pathways map. Both models are described in detail below:

5.1 Profiling Model

Fig. 7.1 shows an example of an individual performance model. In this example, we see that the individual has outstanding reviews, verifications and submissions, however, the tenureship is under the threshold. Based on the classification in Table 7.1, we classify this individual as an active member (SVR). This model is used in Chapter 8.

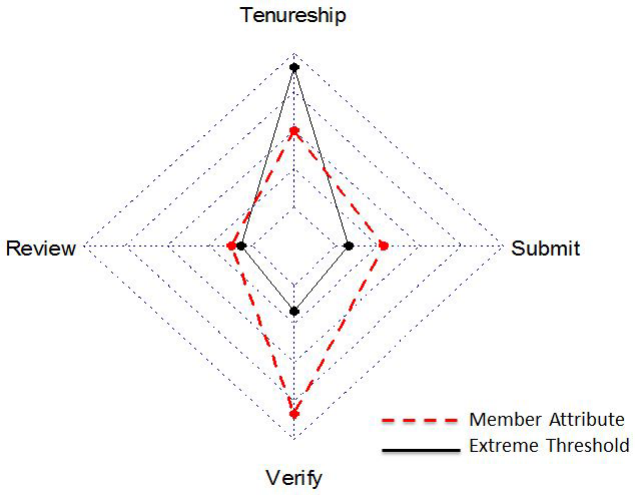


Figure 7.1: Illustration of our profiling using a radar chart.

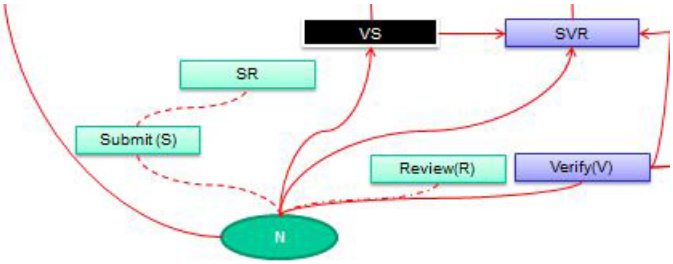


Figure 7.2: Partial representation of a career pathways map.

5.2 Career Pathways Model

Fig. 7.2 is a partial representation of the career pathways map. Each transition is represented by a connected line. Usually the transition is in one direction, however, the dotted line represents if there have been movement in two directions between member types. For instance in Fig. 7.2, there are cases where a member may have become a reviewer (R) before returning to being a non-expert (N). The member type coloured in black represents a member state that currently has no members. Verifier types are coloured in purple. N is the starting point for the map. This model is used in Chapter 9.

6. Contributions

The main contributions and findings of Part III (Chapters 8 and 9) can be summarized as follows:

Peer Review Member Profiling. Profiling and categorised Android members based on contributor activity using our approach.

OSS Health. Identified two possible member types (senior and senior verifiers) as indicators for the health of an OSS project.

Identifying expertise. The identification of hidden/potential experts. Using our metrics, we were able to identify new members with high activity levels.

Visibility of OSS member social standing. Using the profiling metrics, members are able to assess their current peer review performances in the community.

Historical Career Mapping. Using the member types, we are able to track the historic member types transitions, thus able to plot career trajectories for aspiring young members.

We believe that our study is an example of how MPA can be utilised to promote the adoption of SPI initiatives such as peer review for software quality assurance.

7. Chapter Summary

In this chapter, we introduce the theory and related works for MPA of the OSS peer review processes. Since our research is solely using the Android project as the case study, we provide a brief overview of this project, the project roles and other terminology used in the studies. Later in the chapter, we present our proposed metrics and models. The profiling model and metrics are used in Chapter 8, while Chapter 9 uses career path map model. Finally, we presented the main contributions of the study.

Chapter 8

Profiling Peer Review Member Types

1. Introduction

In the previous chapter, we introduced the models and metrics for member profiling. In this chapter, we aim to apply our approach to profile and categorise OSS members based on their historical activities.

To guide our research, we formulated the following research questions:

- **RQ1.** Can we identify and categorise hidden experts?
- **RQ2.** Can we identify inactive members or members with declining interest in the project?
- **RQ3.** Does our expert classification provide practical expert classifications?
- **RQ4.** Are we able to identify differences in patch and processes for the different member types?

2. Results

Using our custom scripts, we extracted 11,512 patch reports over a 38 month period (2008/10/21 - 2012/01/27). During this time, AOSP recorded 1,040 members.

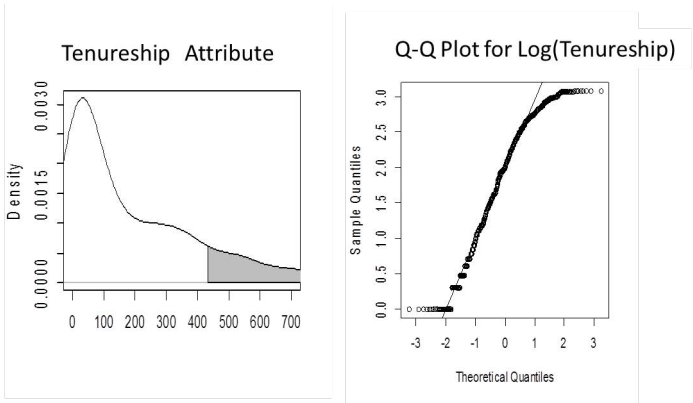


Figure 8.1: The figure illustrates comparison between the uncategorised and the member types of AOSP.

2.1 Extreme threshold evaluation

The pareto principle can be expressed mathematically as a power-law or log-normal distribution. Log normal distributions are characterized for having only positive values and are skewed with long tails. Moreover, they must be log-normally distributed [1] . As an example, we show in Fig.8.1(a) the skewed distribution of one of our attributes, tenureship. The shaded area represents the seniors (extreme thresholds of tenureship) of AOSP. To prove our attributes were log normal distributed, we use Q-Q probability plots (if linear), as shown in Fig. 8.1(b).

We verified the pareto principle for all metrics before calculating the extreme thresholds. Results are shown in Table 8.1. It is interesting that the verify threshold is 0%, this validates that verfiers are automatically extreme members of the project. In Table 8.1, it is shown that members must have at least 12 days of tenureship before considered eligible for classification.

2.2 Member type analysis

As seen in Fig. 8.2, there are considerably more uncategorised members (917 members) in comparison to member types (314 members). The solid line in the

[1] a random variable X is log-normally distributed if log(X) is normally distributed

Table 8.1: Expert Thresholds for AOSP

Tenureship (Days)	Submit Rate (DailyRate)	Review Rate (DailyRate)	Verify Rate (DailyRate)	Eligibility (Days)
430.5	0.100	0.121	0	12

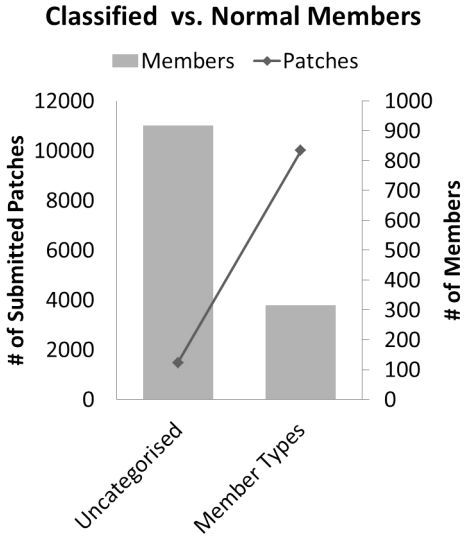


Figure 8.2: The figure illustrates comparison between the uncategorised and the member types of AOSP. Solid line represents the submitted patches by the member types

figure represents the number of patches submitted by the corresponding member types. We can see that a total of 10,050 patch reports were submitted by the member types as opposed to 1,582 patch reports submitted by non-categorised members. This illustrates that experts, even though a minority, contribute the most to the project. This is consistent of the OSS onion model [33], in which with the minority of inner layers taking more leading and contributing roles than the outer layers.

In Fig. 8.3, we grouped together the expert types (with verifier related types as shown in Table 7.1) against the non-verifier categories. Consistent with the onion model, verifiers have more patch contributions.

In Fig. 8.4, we take a closer look at the expert types introduced in this paper

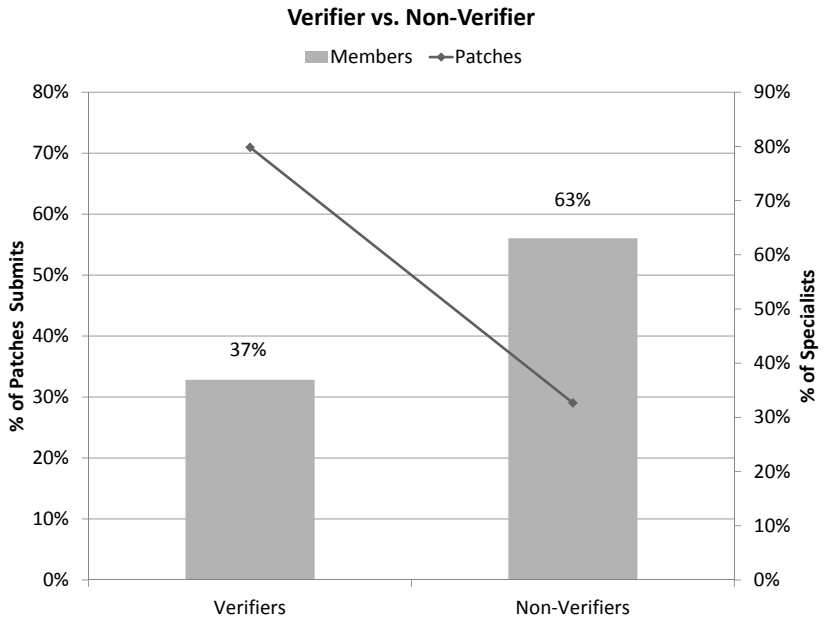


Figure 8.3: The figure illustrates that in AOSP, there are more non-verifiers than verifiers, however more patches are submitted by verifiers. Solid line represents the patches submitted by the corresponding members.

(Table 7.1). Out of the 8 expert types, four are not represented (expert verifier, senior verifier, senior verifying submitter and verifying submitter). This suggests that maybe the verification attribute could be strongly correlated with the other metrics. Table 8.2 shows the results of the pearson correlation tests between the three attributes. Between member types (shown in brackets), there is very strong correlation (0.9) between verify rates and review rates. This indicates that a verifier's takes part in as much reviews as verifications. The relationships between review activities are still not fully understood and could be another interesting avenue for future work.

As seen in Fig. 8.4, the two top member types possess the tenureship attribute (seniors with 30% and senior verifiers with 18.2%). From these results we deduce that senior members could be associated with the project maturity. It would be interesting to apply our approach to a less mature project. As for an indication of the current health of the OSS project, we would need to progressively monitor the progressive growth of senior types over an extended period of time. This will

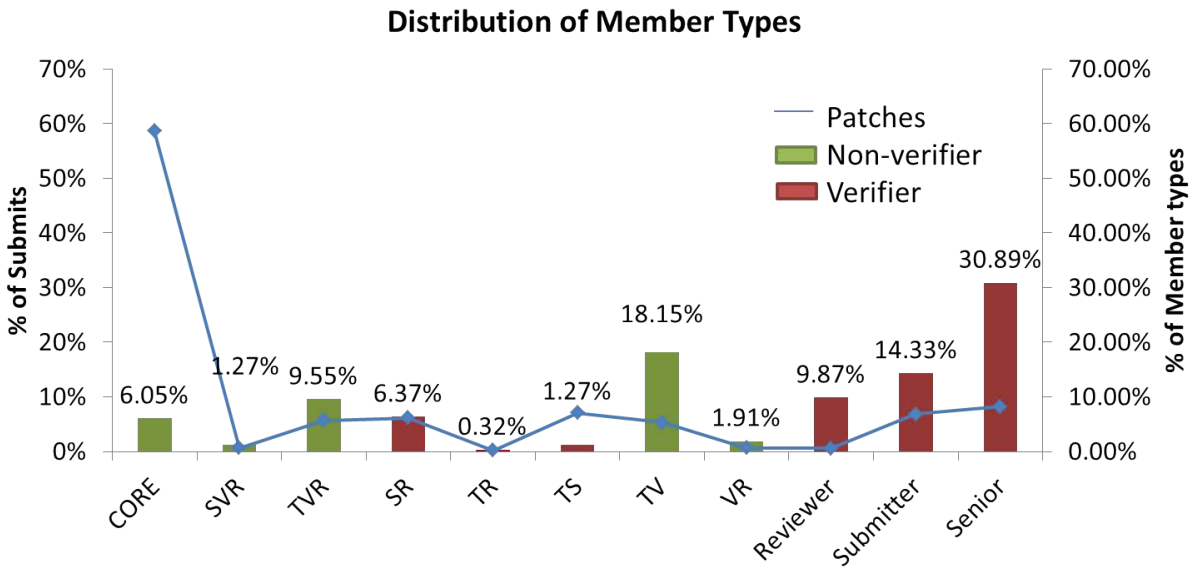


Figure 8.4: This figure shows the distribution of expert types. Please note that four expert types had 0 members at this point in time.

Table 8.2: Pearson Cor. Matrix: T= Tenureship, S=Submit, R=Review, V=Verify,(Verifier's Cor. in brackets)

	S	R	V
S	-	0.063(0.36)	0.026(0.32)
R	0.063(0.36)	-	0.31(0.90)
V	0.026(0.32)	0.31(0.90)	-

be considered for future work.

Finally, core members (with all four extreme attributes) who are considered the most skilled experts with the highest patch submissions, only account for 6.1% of the member types. From a newbie viewpoint, the ultimate career goal would be to reach this type.

2.3 AOSP potential experts

One of the main goals of our research was to identify hidden or potential experts. We approached this using the verifier types. Fig. 8.5 illustrates the identification

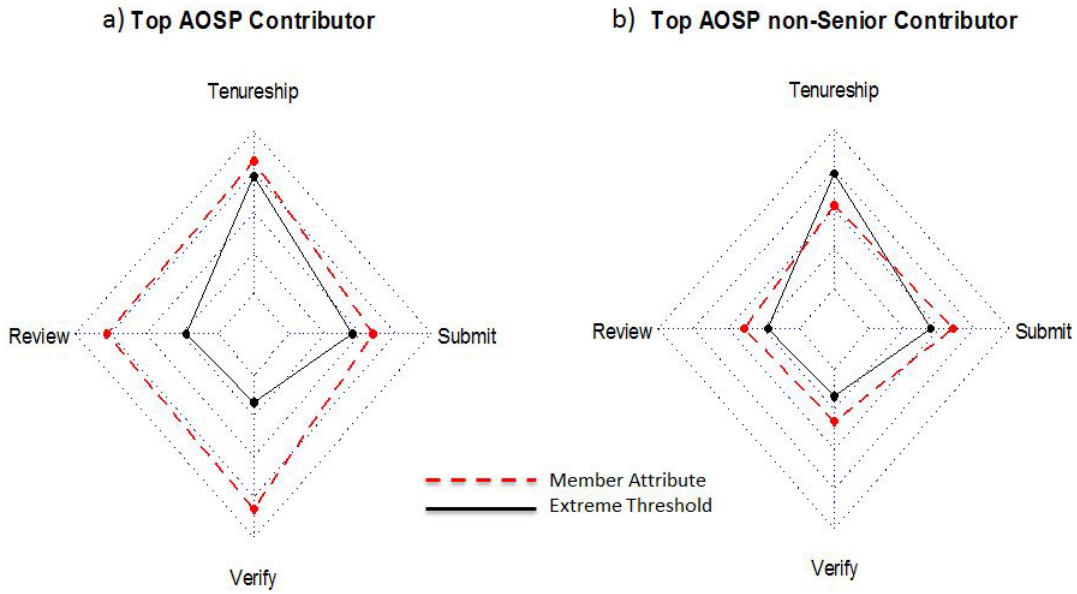


Figure 8.5: Example showing the performances of a) top AOSP contributor and b) a hidden/potential expert (extreme activity although not a senior contributor)

of the hidden experts. In Fig. 8.5(a), the profile of the contributor with the highest attribute scores is shown. It can be observed that the levels of reviews and verifications are particularly high. On the other hand, Fig. 8.5(b) represents the profile of a possible potential hidden expert. This is because this contributor also has extreme activity levels, however, is not considered a senior according to our thresholds. We find that this contributor has only been a contributor for 43 days, but has been very active, verifying 42 patch reports, submitting 21 patches and reviewing an additional 41 patch reports. This user could possibly be a newly employed google developer hired for the Android project, however this claim is not validated. Validation of our results is to be approached in future work.

2.4 Member types properties

As preliminary results, we used 8 metrics to measure the different properties of the member types. These metrics are described in Appendix B. Boxplots comparisons of the median distributions for member types can be located in Appendix C. Below is a summary of the results:

- Senior submitters(TS) and submitters(S) had the longest process review times. Experts that possessed the reviewers(R) attribute had the most comments on their patches

- Core members had the most merged patches. However, along with non-experts and submitters they had also the highest abandoned patches. Core members tend to submit patches that do not change the size of the code (edits), whereas experts with the verifying (V) attribute tend to submit the larger code changes. Additionally, these verifying experts that have not yet possessed the senior status (T) tend to submit more complex patches, affecting a wider number of files (V, VR, SVR).

The results suggest that each expert type has a unique contribution to the project. We believe these insights may assist peer review members plan their future performances to aspire become a certain expert type other than a core member.

3. Discussion

We view our results as the initial steps towards profiling and categorising different member types. Moreover, we also were interested to identify hidden or potential experts. Our initial results indicate that this can be possibly achieved, however validation of our results with the actual project members is required. Thus, *in response to RQ1, results suggest that our member types are able to identify these experts.*

In Fig. 8.4, seniors and senior verifiers compose of most (30%) member types. We suspect that both member types could possibly include inactive or members with declining interest in the project. Results are inconclusive, however, monitoring both these members types could provide useful towards gauging OSS health. Therefore, *in response to RQ2, although we have not validated the categorisation of inactive members, we believe our metrics could be useful OSS health indicators.*

Our metrics generated 8 possible expert types, however, as seen in Fig. 8.4 four expert types are not represented. Results suggest a strong correlation between reviews and verification. Both seniors and senior verifiers types could provide

insights into OSS health.

According to the Android documentation, verifiers are "invited members that have submitted significant amount of high-quality code to the project and demonstrated their design skills and have made significant technical contributions to the project ". Therefore, comparing and understanding their patch review process and code properties could help aspiring members towards the ultimate goal of being a core member type. Individual profiling also helps a young member track and assess her current performance.

Taking all these points into account, *in response to RQ3, results suggest that our metrics and the proposed member types do have practical applications, however, more work is needed to fully understand all of the derived types.*

Finally, our results gave some insights into the properties of each member types. We can summarise that the tenured members tend to submit much smaller and specialised patches as opposed to the aspiring young verifying expert. Also, experts with high reviewing attributes attract attention to their patches then submitters or senior submitters. Much validation is needed, however *in response to RQ4, the difference of patch and process between member types do provide some insights expert type properties.*

4. Threats to Validity

Currently our thresholds are based on the 20-80% pareto rule. This is only used as an example of a set baseline, however in its defence, as seen in Fig. 8.2, Fig. 8.3 and 8.4 the member types, verifiers and the ratio of the number of core members to their submitted patches consistently show that most of the activities are performed by a minority. We believe that the pareto rules could be substituted for other techniques such as Tukey's outlier equation [49].

We identified that the accuracy and validation of our results with the real world is a threat. Specifically the issue of *email aliasing*. By using semi-manual processes of cross-checking the username, name and email address for duplicates, we are confident of our contributors list. As future work, we would like to look at the histories of the Android members, enabling validation of our results.

Our next step is to apply our approach with other similar projects to generalise

our results. We believe that with more projects, we should be able to redefine our thresholds and better understand our expert types.

5. Chapter Summary and Future Work

In summary, the study in this chapter served as a proof of concept for our profiling member types. Using this models we intend to locate experts for certain locations of components and knowledgeable areas of the system. Also, we would like to track inactive or members with declining interest so we can devise counter strategies. Finally, we would like to identify potential 'rising stars', providing this career advancement pathways modelled on current experts.

There are still outstanding issues for future work, including the following:

- validation of results with more real world case studies.
- further investigation of the senior and senior verifiers types at different stages of an OSS project.
- investigation of peer review processes and code patches between expert (verifier) and non-expert types.
- improvement of the expert thresholds by exploring validation.

Currently, we only focus on reviewer profiling, assessing current performance and thier standing in the project. Our overall goal is towards development of an OSS expert recommendation system.

Chapter 9

Career Trajectory for Peer Review Members

1. Introduction

In this study, we present our approach that uses our proposed profiling and career path modelling. Our approach is based on the Goal/Question/Metric (GQM) paradigm first coined by Basili [16]. The previous chapterm our work [60] proposed peer review profiling metrics to categorise *member types* based on review activity levels. Extending on this work, we profile members and then generate career-pathways based on member's transitions of the member types.

We used the Android patch peer review community to test our methodology. Our method traced a member's review activities over a 38 month period with 13 intervals. We used two case scenarios to simulate the use of our approach with success. Results proved the feasibility of our approach, suggesting improvements and possible applications of our approach.

2. Proposed Approach

2.1 Goal/Question/Metric (GQM)

Fig. 9.1 presents the overall approach. As shown, our methodology is based on Basili's Goal/Question/Metric (GQM) paradigm [16]. This is presented below:

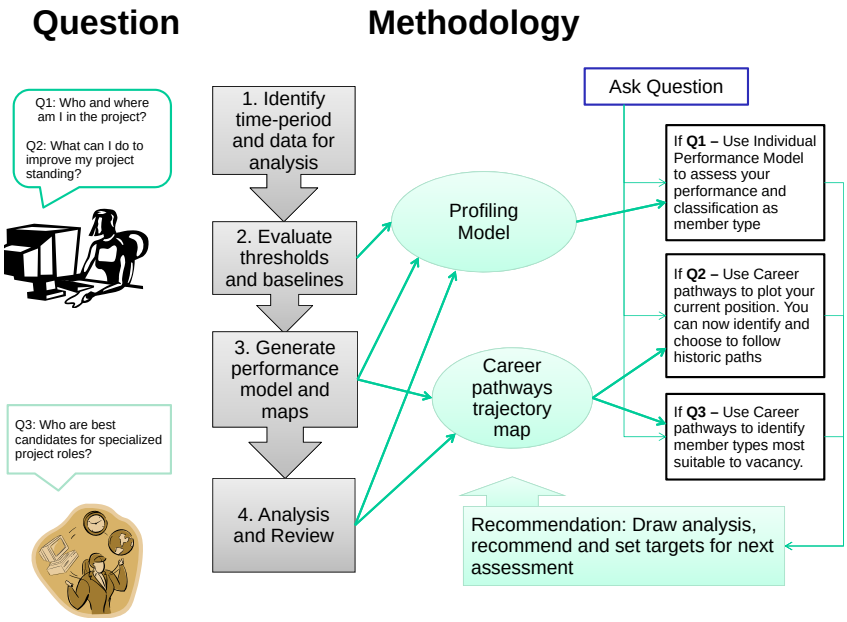


Figure 9.1: Overview of our Approach. First we have our question, then we formed of methodology that uses the two empirical models.

- **Conceptual Goal**. Our goal is to provide visibility and awareness of an OSS contributor's social status within the peer review community.

- **Question**. Our questions are based from two viewpoints. First, from the member stance, we ask:
  1.) Who and where am I in the project?
  2.) What can I do to improve my project standing?
  Finally, from management point of view, management may ask:
  3.) Who are the best candidates for specialized project roles?

- **Quantitative Metrics and Models**. Instead of the traditional metrics, in this stage we define our methodology and two quantitative models (profiling model and career pathways trajectory map) to address our questions.

Table 9.1: Time-Frame Intervals

Date	Dec-08	Mar-09	Jun-09	Sep-09	Dec-09	Mar-10	Jun-10
Interval	1	2	3	4	5	6	7
Date	Sep-10	Dec-10	Mar-11	Jun-11	Sep-11	Dec-11	
Interval	8	9	10	11	12	13	

2.2 Methodology

In this section we describe the methodology of our approach. As shown in Fig. 9.1, there four steps in our method:

1. **Identify time-period and datasets.** Firstly we identify the time-period for the performance evaluation. Quantitative data is collected of all review activities for the individual.

2. **Evaluate thresholds and baselines.** Using the threshold attributes, we measure the individual performance against the outstanding individuals. Based on the performance levels, we are now able to classify the individual into the member types in Table 7.1.

3. **Generate empirical models.** Based on the data collected and the thresholds, we are now able to generate the profiling model and career pathways map defined in Chapter 7.

4. **Analysis and Review.** We propose that aspiring members can use the historic pathways of current member types to plan their career trajectories. From a management standpoint, management can choose suitable candidates from the member types.

3. Application: Analysis by Evolution

For our analysis, we study the evolution of members during a time-frame. As shown in Table 9.1, we used a time-frame of 3 months for 13 intervals. Using our custom scripts, we extracted 11,512 patch reports over a 38 month period (2008/10/21 - 2012/01/27). During this time, AOSP recorded 2,072 members.

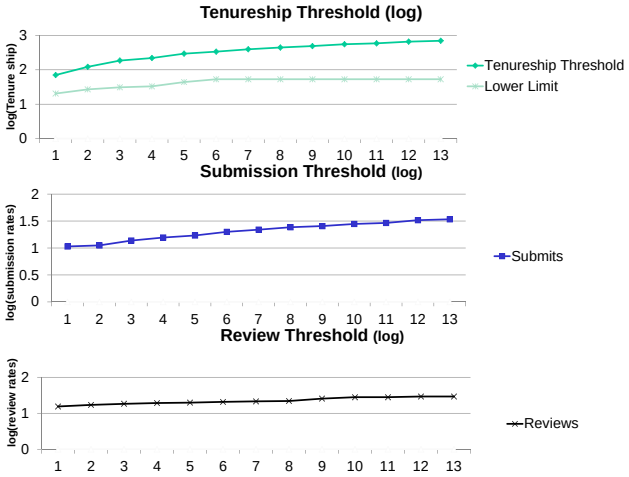


Figure 9.2: The different thresholds taken during the 13 intervals for the tenureship, patch submission and review activity.

We used relational databases to analyse our data and the R Tool [1] for statistical analysis and to generate our profiling charts [2] . Our compiled dataset is readily available online for download[3] .

The rest of the section is divided into two parts. First, in section 3.1 the threshold evaluation and member types transitions during the time-frame are presented. Later in section 3.2, we then present two case scenarios to which we apply our method.

3.1 Thresholds and Member Types

Similar to the case study in Chapter 8, we used the pareto principle as our threshold to identify review activities that exceeded 80% of the population. We can see in Fig. 9.2, that the thresholds have a steady rise over the time-frame. We found a strong correlation between all thresholds over time, especially with the submit rate threshold *(pearson r=0.98)*. The constant rise suggests that maintaining extreme attributes requires activity review participation. Table 9.2 shows the correlation between review activity types taken at the final interval

[1] http://www.r-project.org/

[2] http://cran.r-project.org/web/packages/fmsb/fmsb.pdf

[3] http://sdlab.naist.jp/reviewmining/

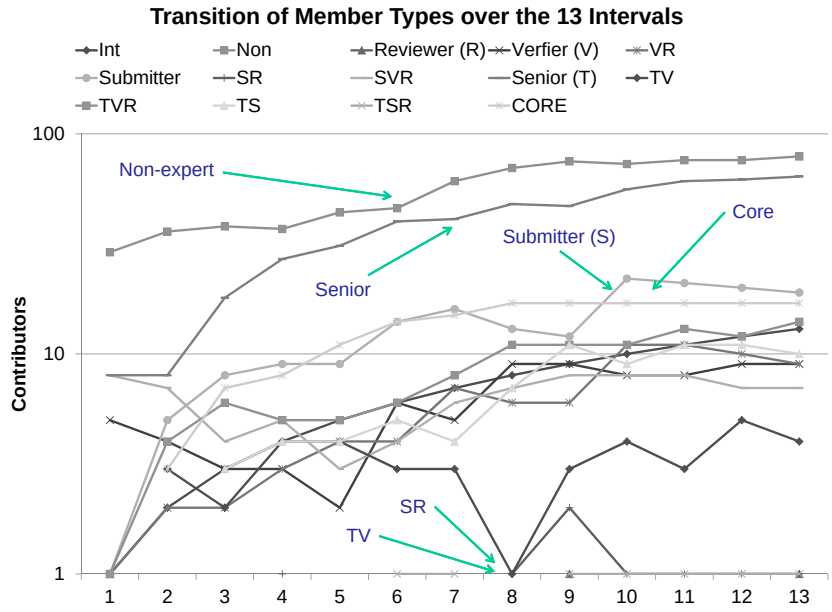


Figure 9.3: Transitions of members between member type classifications during the 13 intervals.

(13). Results suggest a very strong correlation between verification and review activities.

Fig. 9.3 shows the transition changes between member types over the time-frame. We can see that both member types senior (T) and non-expert (N) show a steady growth, while other types such as TV and SR had no members at intervals 8 and 10. Other groups such as submitter (S) and core members (TSVR) look to have reached a peak and then steadily remain at that level. Such observations could be important in understanding the roles of these member types in the community.

3.2 Case Scenarios

To test our method, we applied our approach to a sample of Android review members. We selected a current core member, known as userid *1000411*. We assume that most contributors would like to become core members at any point in their review life. In the second scenario, we assume that 1000411 has decided to

Table 9.2: Pearson Corr. Matrix at Interval 13: T= Tenureship, S=Submit, R=Review

	T	S	V	R
T	-	0.16	0.23	0.24
S	0.16	-	0.50	0.48
V	0.23	0.50	-	0.88
R	0.24	0.48	0.88	-

leave the community, therefore project leads are faced with filling in that vacancy. Scenario one is addressed by Q1 and Q2, while scenario two by Q3.

Q1. Who and where am I in the project?

Using our profiling model, we are able to trace the performances of user 1000411 over the 13 intervals. As shown in Fig 9.4, we can see this user started out as member type *N*, later becoming a verifier type *VR* and *SVR*. Eventually, the user became a core member (*TSVR*). 1000411 first increased rates in reviewing and verifications, then submitting and finally gaining tenureship to become a core member.

Additionally, as shown in Fig. 9.5, we can see a more detailed overview of the career pathways map for 1000411. 1000411 underwent three member type transitions to reach the core member status, however, did spend a year (time-frame 2-5) as a VR member before progressing in her career. These models show that at any period in time, we are able to dynamically profile and generate a historic career path for any user.

Q2. What can I do to improve my project standing?

To improve the current standings, we assume that the best approach is to learn from experience, which is following the footsteps of role models which in this case are core members. Fig. 9.6 shows all the career paths of the current 17 core members. From the map and corresponding table, we can see that most core members became SVR types before reaching core member status. 1000411

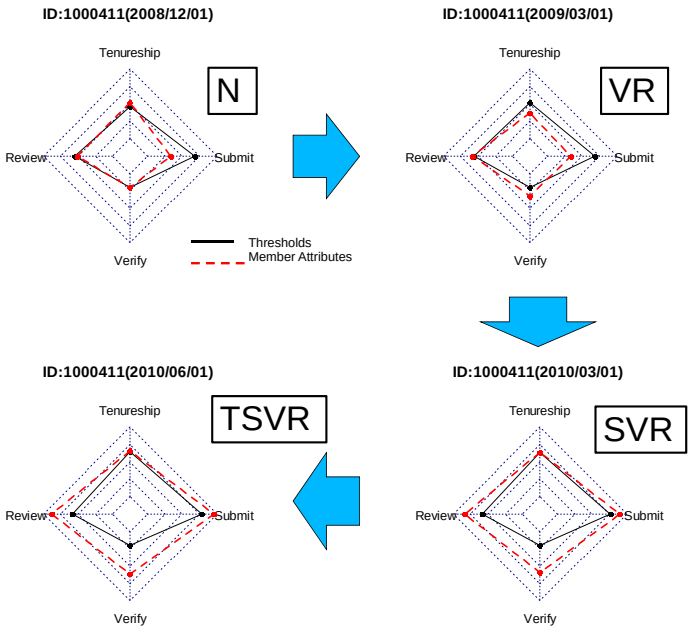


Figure 9.4: Snapshots of the changing profiles of 1000411. Each snapshot is taken right after a member type transition (reads top left to right, then right to left at the bottom.

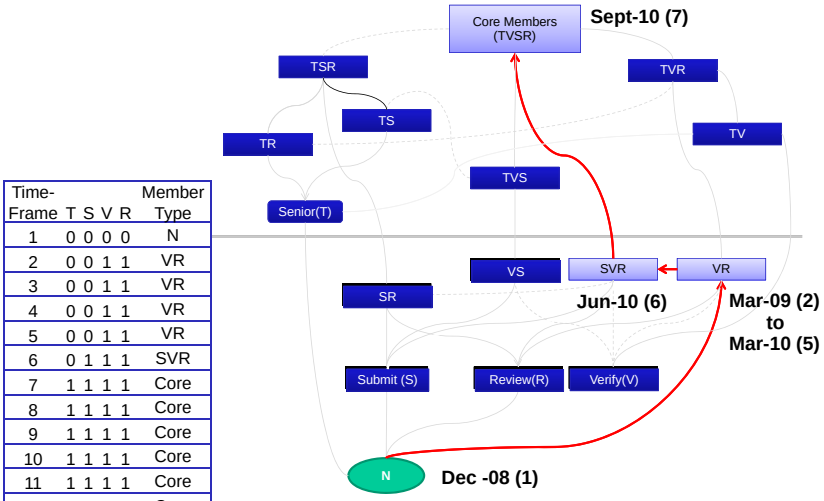
is currently a core member, so we cannot suggest any improvements. However if she asked the same question back when she was a VR we could suggest that she raise her submission levels to become a SVR type. As seen in Fig. 9.6, historically more core members took this same pathway through the SVR member type.

Furthermore, Fig. 9.7 shows all the career pathways of all types over the 13 intervals. As shown in this career map, three of the member types currently have no members (TVS, TR, VS). It is interesting to note that R, SR and TSR member types have no higher transition in member types, as members seem to return to the previous member type.

Q3. Who are the best candidates for specialized project roles?

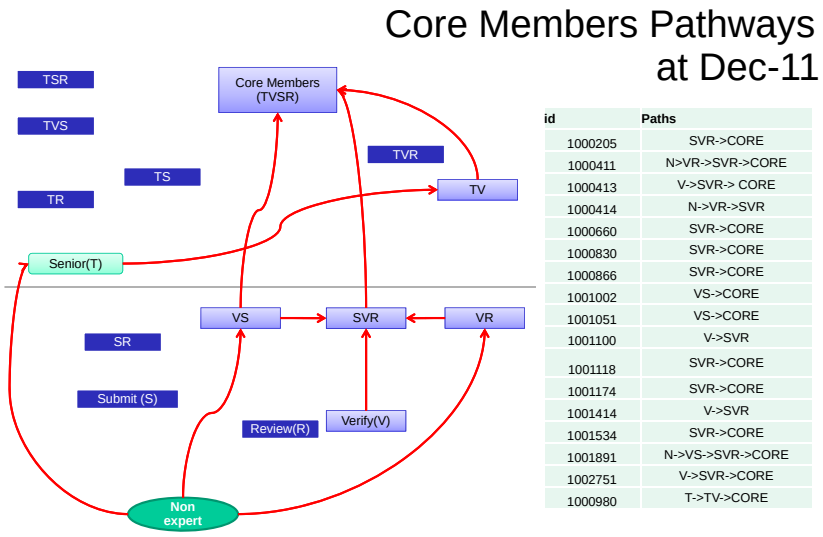
It is assumed that a vacancy in a project occurs when either because of overbearing workloads or if a core member leaves. In the latter case, we can utilise

Example: User ID:1000411



Time-Frame	T	S	V	R	Member Type
1	0	0	0	0	N
2	0	0	1	1	VR
3	0	0	1	1	VR
4	0	0	1	1	VR
5	0	0	1	1	VR
6	0	1	1	1	SVR
7	1	1	1	1	Core
8	1	1	1	1	Core
9	1	1	1	1	Core
10	1	1	1	1	Core
11	1	1	1	1	Core
12	1	1	1	1	Core
13	1	1	1	1	Core

Figure 9.5: Illustrates the career map for 1000411 over the 13 intervals.



id	Paths
1000205	SVR->CORE
1000411	N>VR->SVR->CORE
1000413	V->SVR-> CORE
1000414	N->VR->SVR
1000660	SVR->CORE
1000830	SVR->CORE
1000866	SVR->CORE
1001002	VS->CORE
1001051	VS->CORE
1001100	V->SVR
1001118	SVR->CORE
1001174	SVR->CORE
1001414	V->SVR
1001534	SVR->CORE
1001891	N->VS->SVR->CORE
1002751	V->SVR->CORE
1000980	T->TV->CORE

Figure 9.6: Historic career pathways map for core members only during the 13 intervals

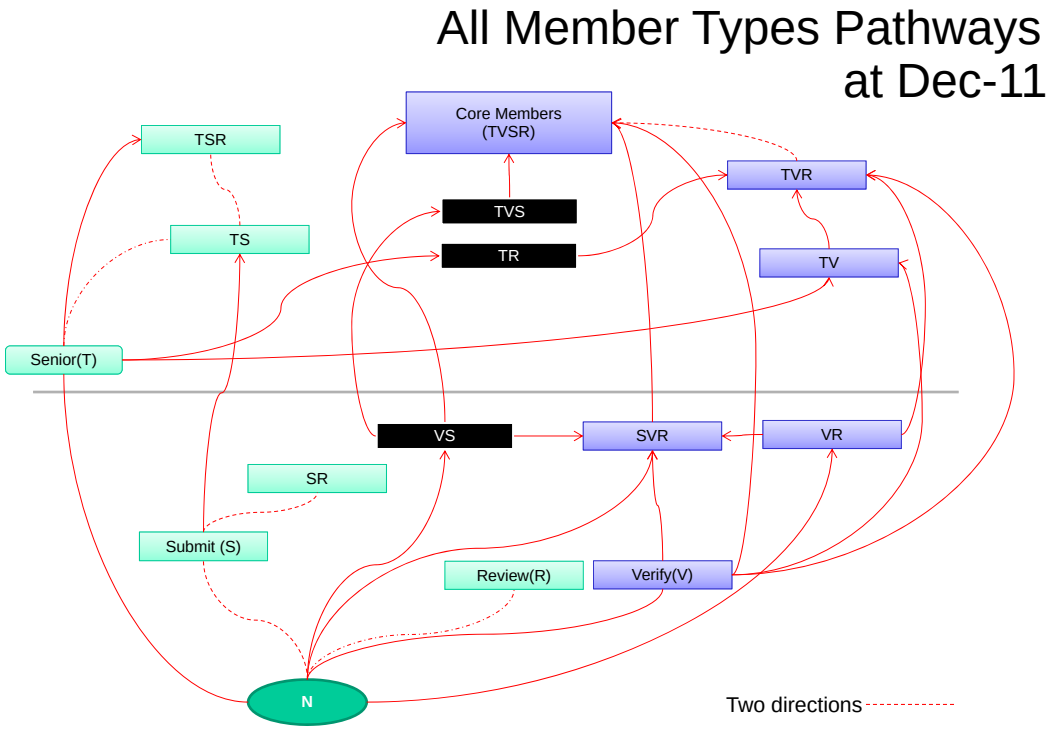


Figure 9.7: Historic career pathways map for all member types during the 13 intervals

our method to find the best candidates by identifying users with similar historical traits. For example if 1000411 was to leave the project, we propose that contributors that experienced a similar career path could be suitable candidates to take on the workload. This could be any current member that is currently a SVR member types as that was the previous member type transition 1000411 has become a core member. Due to computation costs, we are unable to provide a concrete answer to the question and regard this as future work.

4. Discussion

The objective of our approach is to provide a visibility of a contributor's current standing within an OSS project and possible career trajectories based on historic

career paths of successful contributors. Our method should only be used as a guideline for career trajectories as attainment of higher social standing is only through project lead selection based on invitation and display of technical skill. By studying the actual patches of known expert member types could give insights into the technical expertise needed to attain that member type.

Other benefits that could arise from this work is understanding how OSS projects are able to sustain a community for its livelihood. For example, OSS projects might require a certain number of member types for its success. Conversely, an increase in other member types might indicate declining interest or leaving members.

The costs and benefits of transitions from one member type to another could be another avenue to research. For example, some member types could provide more attention from reviewers or project leads while others could be stepping stones into best positions for promotion. Identification of member types saturation points, proximity to other types and features such as the review processes and patches associated with of different member types could provide insights into the different qualities of each member types.

Member types could be used to identify expertise and knowledge related to specific aspects of the project. We suggest that career path of members could be greatly influenced by the frequency and duration of transitions of the member types. These metrics could provide additional experience metrics. The investigation of which member types enjoys shorter review times, which members types draws more attention of management or a what types are simply stepping stones into greater roles could be prove interesting future work. For management, we propose our method to provide more experience metrics as well as identify similar aspiring members. Other uses could be to assist with the trigation of patches to suitable reviewers and verifiers.

Currently we use radar charts to visualize our profiling model. From a similar perspective, the current career mapping is performed manually. As mentioned in the related work, career path and sequence analysis have been used to analyse state transitions [70]. Data mining analysis techniques such as Markov chains or self-organising maps will be explored as future work.

Another important aspect of this work is the validation of our model with the

real world. At this early stage of the research we are more concerned with the proof of concept. We plan to validate our results with actual members. Also, we would like to investigate the impact of real world events such as new version releases or departure of a core member on our models.

5. Threats to Validity

5.1 Internal

Currently our thresholds are based on the 20-80% pareto rule. We believe that the flexibility to adjust the threshold could be favourable for tailoring our approach. For instance, the pareto rules could be substituted for other techniques such as Tukey's outlier equation [49].

In this study, we chose a three month time frame interval. We believe that this was sufficient to capture member type general transitions, however, there were cases as seen in Fig. 9.6, where contributors gained more attributes within an interval. Further investigation is needed to refine the time-frame to capture member type transitions.

We identified that the accuracy and validation of our data is a threat. Specifically the issue of email aliasing, which was also encountered by Bird [24]. By using semi-manual processes of cross-checking the username, name and email address for duplicates, we are confident of our contributors list. As future work, we would like to look at the histories of the Android members, enabling validation of our results.

5.2 External

We believe our method captures the basic review activities, thus can be used for other real world projects. We believe that the flexibility of the thresholds should make our method applicable to many different types of projects that employ a patch peer review system. Our next step is to apply our approach with other similar projects to generalise our results.

6. Chapter Summary and Future Work

We foresee many benefits into the profiling and historic analysis of of OSS peer review careers. Currently, we only focus on reviewer profiling and historic career path generations. Our end goal is towards better understanding of career advancement and individual identity in an OSS setting. Our motivations stem from expert finding, knowledge management and understand how to combat the decline of interest in OSS projects. Finally, the identification of potential "rising stars" to distribute workloads as well as fill vacancies of specialized project roles.

As a feasibility study, we believe our work opens many avenues for research. There are still outstanding issues for future work, including the following:

- validation of results with real world activities, such as comparing real life events with changes/transitions within the models.
- validation of our approach with other real world projects for generalisation.
- investigation of the specialized attributes of member types.
- improvement by investigating other techniques, for instance the thresholds and automation of career path mapping.

We believe that due to the recent improvements of data repositories mining tools and techniques, this research is timely and will be able to give quantitative insights to experience within OSS projects.

Part IV

Synopsis

Chapter 10

Conclusions

Recent developments in software repositories and mining techniques have given recent rise to MSR studies. In this dissertation, we would like to apply MSR techniques to other sub-fields of software engineering such as Software Process Improvement (SPI) and Software Maintenance (SM). This thesis makes the following contributions to this area.

- **Literature Review of Quantitative SPI and SM.** 44 papers were selected from 7 premium journals and conferences. 66% of SPI papers originated from the industry as opposed to only 10% from OSS and 7% SPI in other sources. This suggests that SPI has more industry contributions. However only 7% of SPI are quantitative (correlation and experimental) studies. These results suggest differences between practitioners and researchers in the software process improvement (SPI). The review suggests there are opportunities for the use of micro process analysis to complement current models and metrics to provide quantitative SPI initiatives. (Chapter 2)

- **Quantitative relationship between maintenance effort and its impact on source code.** In this new approach to complement the current SPI assessment models, we propose quantitative program sliced metrics to measure change impact. Together with a quantitative expression of maintenance effort, we provide an quantitative proof of Lehman's second law of software evolution: *"As a system evolves its complexity increases unless*

work is done to maintain or reduce it" [66]. Our study was performed on three OSS projects and a closed experiment to demonstrate general application. (Part II)

The benefits of this research is twofold. First, identification of high maintenance efforts as well as the affected code, especially those with relatively high CC and FC characteristics, could be beneficial from a source code maintenance point of view. These code portions could be candidates for source code maintenance activities such as refactoring, code reviews and code inspections. Also, the affected code could be marked as complex, so that a developer can take extra care when modification of these portions of code is required. Secondly, from a project team standpoint, the ideal scenario is to assign the best developers and resources to the code changes that require the most effort. The team can be reorganized so that the most experienced developers and resources are assigned to *'high maintenance effort'* portions of code.

In this dissertation, we were able to prove the use of MPA for the expression of maintenance effort. In terms of practical application to current SPI models this opens up a new field of investigation. A possible avenue could be the identification of controllable factors at the micro level to help improve micro processes.

There are other secondary directions for future work. Firstly, application of our approach in an industrial setting. Secondly, one of the threats of using program slicing for change impact is its vulnerability at large scale use. Research using different techniques such as heap slicing across different language platforms could be another avenue for future study. In addition, the accuracy between backward and forward slices could help us detect defects quicker.Finally, our novel quantitative measurement of maintenance effort could be used in other applications.

- **Peer Review Members Profiling and Career Trajectory in OSS Projects.** Peer review member profiling for expertise identification can lead to a more efficient and higher quality review process. We propose two empirical models. First is a profiling type to categorise members based

on contribution. This provides a visibility of a member's current standing within a project. Second, we propose a career trajectory models based on the historic transition of member types. (Part III)

Since OSS members are motivated purely by self-interest [64], profiling could help in two ways. Firstly from a management point of view, we could classify inactive members. An increase in this category of members could be an indicator of the degrading health of the project. Secondly, by classifying expert types, we can study their review habits and activities. Using MPA we proposed a profiling and career trajectory models to encourage member's participation and career advancement.

Application of our models to other real world projects is seen as one of the many future directions. Another avenue could be the detailed investigation of member types properties, their benefits and impact to the sustainability of the project. Also in our profile models, the pareto rule was used as a threshold rule. Further testing and calibration is needed to determine the best threshold method. For our career pathways model, at this stage it is semi-automatic. Markov chains and other probabilistic models and techniques could be implemented to automate and improve the model.

Our techniques and approaches prove the application of MPA to software maintenance activities such as bug fixing and peer reviews. On one hand, we envision that MPA methods will open avenues of more quantitative measurement to complement the current SPI models. This would be beneficial for smaller organisations as it saves assessment and implementation costs. Also, specific phases such as the maintenance phase can be targeted.

On the other hand, the emergence of OSS has seen the adoption of current SPI strategies. Since OSS is developer driven, it would be appropriate for the application of MPA for SPI. Profiling and identification of members are the first step toward finding and recommendation of expertise.

The maturity of tools and technologies has sparked new life for quantitative process analysis and modelling at the micro level. Our main goal is that our proposed micro models and techniques are used to inform and complement current models, thus provides an *"added dimension"* to SPI . We end with this quote, *"even at the micro level, it's the processes that define the product"*.

Part V

Appendix

Appendix

Appendix A - Tutorial Slides

Software Maintenance Effort

C++ Coding Experiment

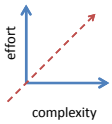
Software Design lab
Iida-ken

Outline

- {STOP} Please fill in PART A -experiment questions!
- Goal
- Case Study: AlignMe
  - Intro
  - System Design
  - Detailed Code inspection
- Maintenance Tasks
  - Task 1 and 2
  - Task 3 and 4
- Conclusion and Feedback

Experiment Goal

- Measure effort for code changes (modifications) with the complexity of portions of code.
- Relationship between effort and source code properties



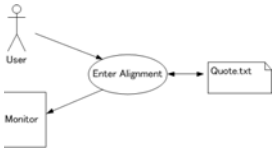
Experiment: Case Study

- **AlignMe** – a simple C++ application

Displays text from a file (quote.txt) and displays in the selected alignment (Left(1) or Right(2)).

- Build and Execute (Single file)
  - Compile: **g++ -o strategy strategy.cpp**
  - Run: **./strategy**
  - Development Environment. (IDE): **Visual Studio 2008**
- For this study, illustrates the strategy design pattern with typical maintenance activities applied

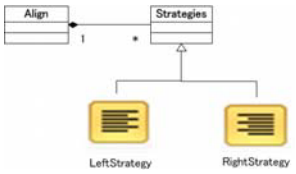
System Design



User selects the desired alignment, system then reads the text from Quote.txt and displays in the desired alignment on the monitor

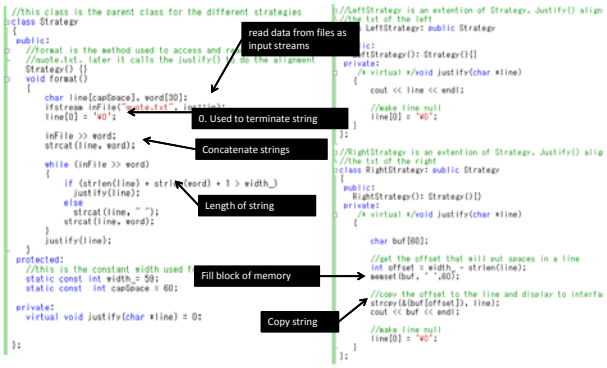
Strategy Design Template is used to:

- Define a *family of algorithms*, encapsulate each one, and make them interchangeable.
- Capture in an interface, bury implementation details in derived classes
- Used in refactoring to make code easier to read and append enhancements

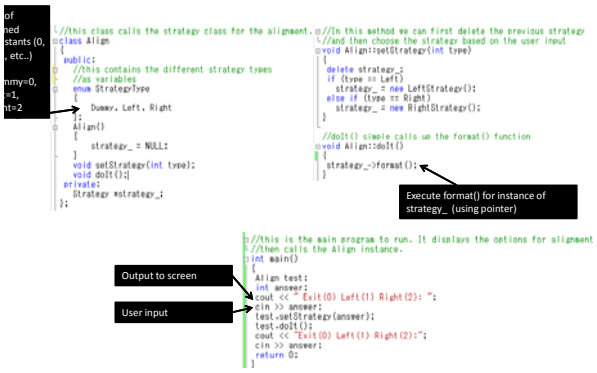


Using the strategy design pattern, left and right strategy classes are derived from the strategies class. The Align class creates an instance of the strategy.

Strategy Implementation



Alignment Class and Main

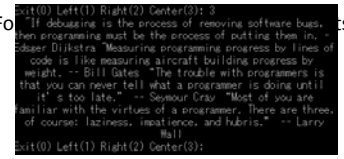


Maintenance Activities

- Three types of Maintenance Activities (tasks):
  - Perfective/Adaptive: Enhancements
    - Task 1 and 2
  - Corrective: Fixing bugs
    - Task 3 and 4

Task 1

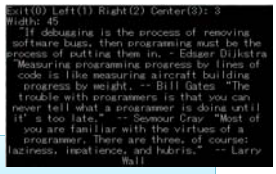
- Allow user to be able to perform center alignment
  - Hint 1. Fo

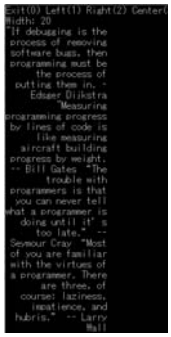


Test Case: implement above screenshot for *center(3)* option

Task 2

- Allow the user to enter a customized width (#of char in line) variable
  - Hint 1. Pass the variables from the main to each strategies





SYNTAX for C++
public:
method(type var): *method/variable*(var){}

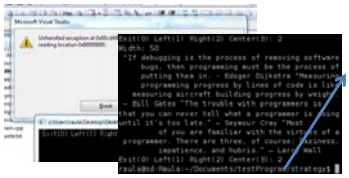
Key variable:
width_

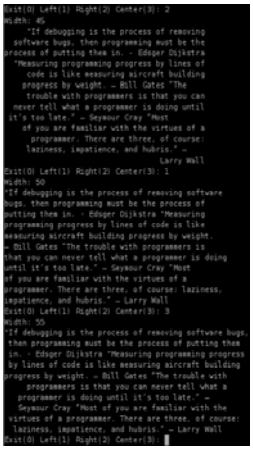
Test Case: Emulate the above screenshot. Enter
- Right/Width=45,

Task 3

A. Fix Exit(0) option.

B. In addition, make the program run until you choose exit.

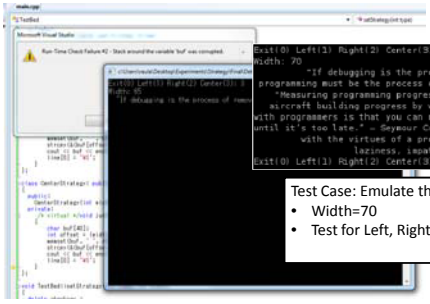




Test Case: Emulate the right screenshot. Enter :
- Right/Width=45,
- Left/Width=50
- Center/Width=55

Task 4

- Make the program able to handle widths of up to 80.



Test Case: Emulate the above screenshot. Enter
- Width=70
- Test for Left, Right and Center cases

Task 1

```
class Align
{
  public:
    enum StrategyType
    {
        Dummy, Left, Right, Center
    };
```

```
class CenterStrategy: public Strategy
{
  public:
    CenterStrategy(): Strategy(){}
  private:
    /* virtual */void justify(char *line)
    {
        char buf[80];
        int offset = (width_ - strlen(line)) / 2;
        memset(buf, ' ', 80);
        strcpy(&(buf[offset]), line);
        cout << buf << endl;
        line[0] = '\0';
    }
};

void Align::setStrategy(int type)
{
  delete strategy_;
  if (type == Left)
    strategy_ = new LeftStrategy();
  else if (type == Right)
    strategy_ = new RightStrategy();
  else if (type == Center)
    strategy_ = new CenterStrategy();
}

void Align::doIt()
{
  strategy_->format();
}

int main()
{
  Align test;
  int answer;
  cout << "Exit(0) Left(1) Right(2) Center(3): ";
  cin >> answer;
```

Task 2

```
int main()
{
  Align test;
  int answer, width;
  cout << "Exit(0) Left(1) Right(2) Center(3): ";
  cin >> answer;
  if(answer)
  {
    cout << "Width: ";
    cin >> width;
    test.setStrategy(answer, width);
    test.doIt();
    cout << "Exit(0) Left(1) Right(2) Center(3): ";
    cin >> answer;
  }
  return 0;
}
```

```
void Align::setStrategy(int type, int width)
{
  delete strategy_;
  if (type == Left)
    strategy_ = new LeftStrategy(width);
  else if (type == Right)
    strategy_ = new RightStrategy(width);
  else if (type == Center)
    strategy_ = new CenterStrategy(width);
}

class Align
{
  public:
    enum StrategyType
    {
    };
    Align()
    {
    }
    void setStrategy(int type, int width);
    void doIt();
  private:
    Strategy *strategy_;
};
```

```
class Strategy
{
  public:
    Strategy(int width): width_(width){}
    void format()
    {
    }
  protected:
    int width_;
  private:
    virtual void justify(char *line) = 0;
};

class LeftStrategy: public Strategy
{
  public:
    LeftStrategy(int width): Strategy(width)
  private:
    /* virtual */void justify(char *line)
    {
    }
};

class RightStrategy: public Strategy
{
  public:
    RightStrategy(int width): Strategy(width
  private:
    /* virtual */void justify(char *line)
    {
    }
};

class CenterStrategy: public Strategy
{
  public:
    CenterStrategy(int width): Strategy(widt
  private:
    /* virtual */void justify(char *line)
    {
```

Task 3

```
//this is the main program to run. It displays the options
//then calls the Align instance.
int main()
{
  Align test;
  int answer, width;
  cout << "Exit(0) Left(1) Right(2) Center(3): ";
  cin >> answer;
  while(answer)
  {
    cout << "Width: ";
    cin >> width;
    test.setStrategy(answer, width);
    test.doIt();
    cout << "Exit(0) Left(1) Right(2) Center(3):";
    cin >> answer;
  }
  return 0;
}
```

Task 4

```
class Strategy
{
  public:
    Strategy(int width): width_(width){}
    void format()
    {
        char line[40], word[20];
        ifstream infile("quote.txt", ios::in);
        line[0] = '\0';
```

```
class RightStrategy: public Strategy
{
  public:
    RightStrategy(int width): Strategy(width){
  private:
    /* virtual */void justify(char *line)
    {
        char buf[80];
        int offset = width_ - strlen(line);
        memset(buf, ' ', 80);
        strcpy(&(buf[offset]), line);
        cout << buf << endl;
        line[0] = '\0';
    }
};

class CenterStrategy: public Strategy
{
  public:
    CenterStrategy(int width): Strategy(width)
  private:
    /* virtual */void justify(char *line)
    {
        char buf[80];
        int offset = (width_ - strlen(line)) /
        memset(buf, ' ', 80);
        strcpy(&(buf[offset]), line);
        cout << buf << endl;
        line[0] = '\0';
    }
};
```

Appendix B - Peer Review Process and Patch Metrics

Peer Review Process Model

Fig. 10.1 illustrates our proposed micro process review process model used in this study. Based on the Android review process[1] , we defined a generic process model. The phases are detailed below:

1. *Patch Submitted.* This is when a patch is submitted by a contributor. We assume this phase is signified by the registration and creation of a new patch review.

2. *Reviewers Assigned.* After submission, reviewers, verifiers and approvers are assigned to the patch for review.

3. *Patch Review.* This process is signified by reviewers response and comments on the code. During this phase there are 3 activities that take place: 3.1) reviewers make comments and suggestions, 3.2) editing and rework of the patch and finally 3.2) testing for verification of the patch . Finally the phase is ended by either acceptance or rejection of the patch.

4. *Patch Review Completed.* Once the patch is either: a) Merged into the source code or b) Abandoned, the patch review process is completed.

Note patch status are related to the review process phases `Open` refers to phases 1-3, with `Merged` or `Abandoned` dependant of the outcome at phase 4. In this study, we focused on abandoned patches, as well comparing to merged patches. We regarded the opened patches as outside the scope of this study.

The process metrics are used to evaluate the review process. From these metrics, we aim to understand *review effort* for a particular patch.

- **Patch Review Duration.** This metric measure the duration in which a patch was reviewed. Unit of measure is the number of days.

[1] http://source.android.com/source/life-of-a-patch.html

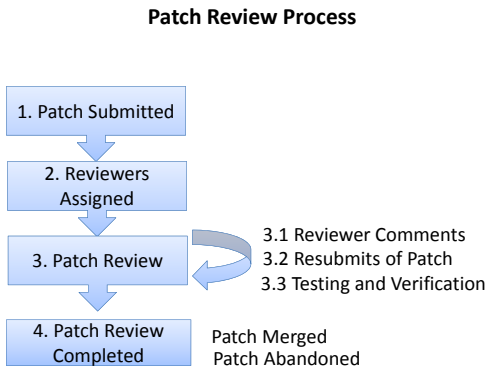


Figure 10.1: OSS Review Process Model.

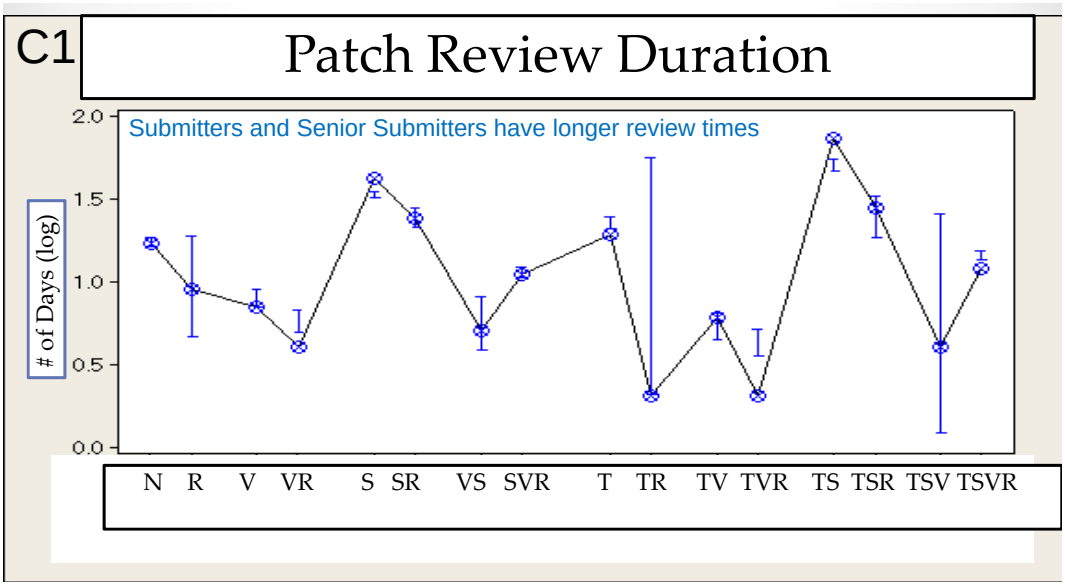
- **Comment Activity.** This metric measures the number of comments during the review phase. These metrics are useful to gain insight into the discussions or responsiveness of reviewers.
- **Merged Patches.** These are the number of merged patches.
- **Abandoned Patches.** These are the number of abandoned patches.

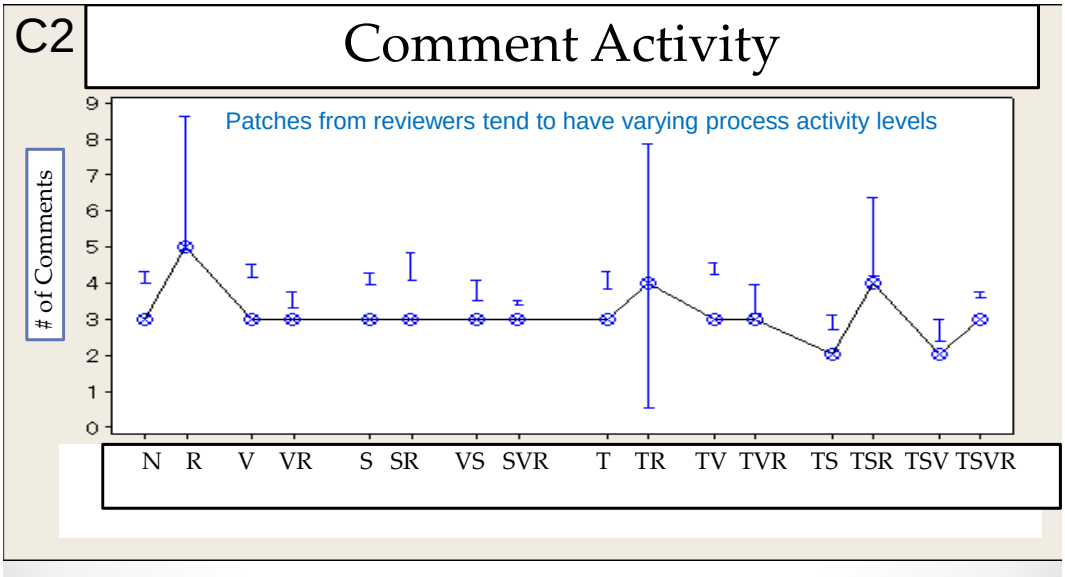
Patch Code Metrics

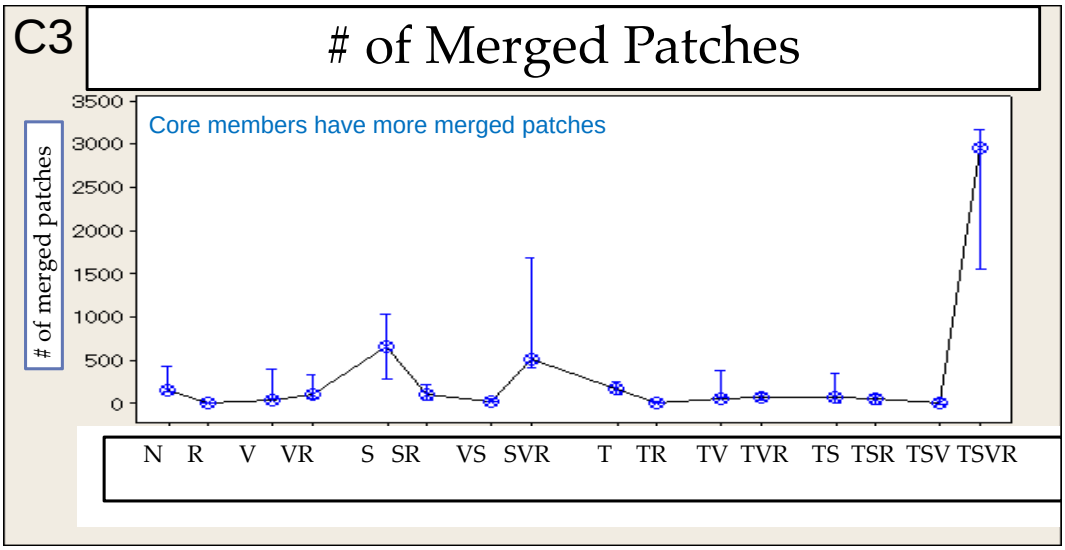
Our proposed patch metrics measure the source code properties of the patch, measuring the size as well as how many files it affects. These are very common code churn metrics [80]. All sizes are measured per Lines of Code.

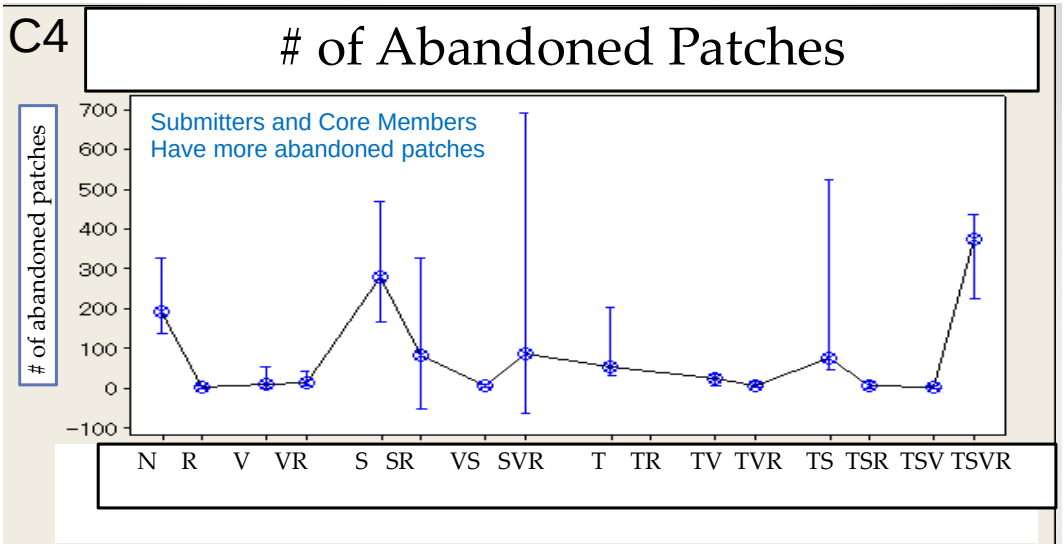
- **Size of deleted patch.** This is a patch that reduces the source code size after merge with source code.
- **Size of merged patch.** This is a patch that will increase the source code size after merge with source code.
- **Edit patches.** An edit patch is a patch that does not add or reduce the size of the source code.
- **Files affected by patch.** Measured in the number of files, we use this metric to measure the coverage and complexity of the patch. For example, a patch may be smaller in size, however may affect many files.

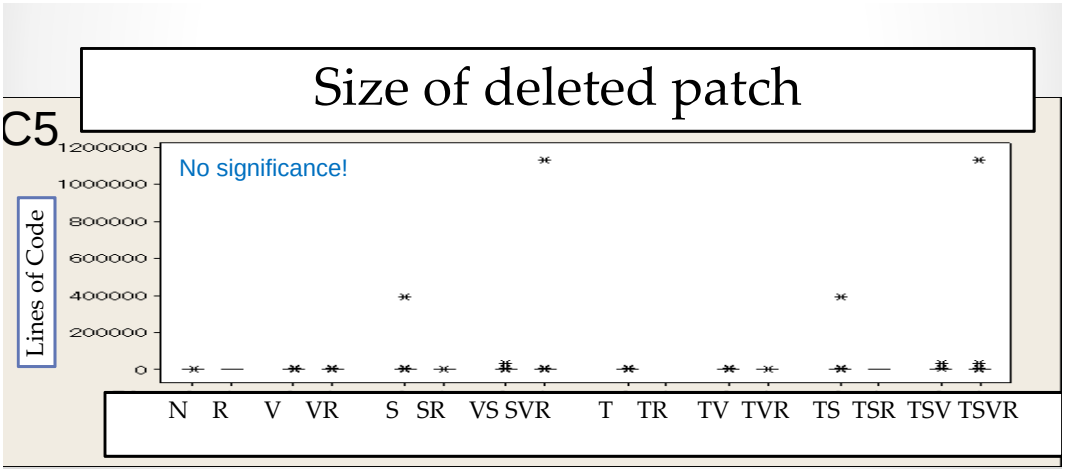
Appendix C - Results of Expert Type Properties

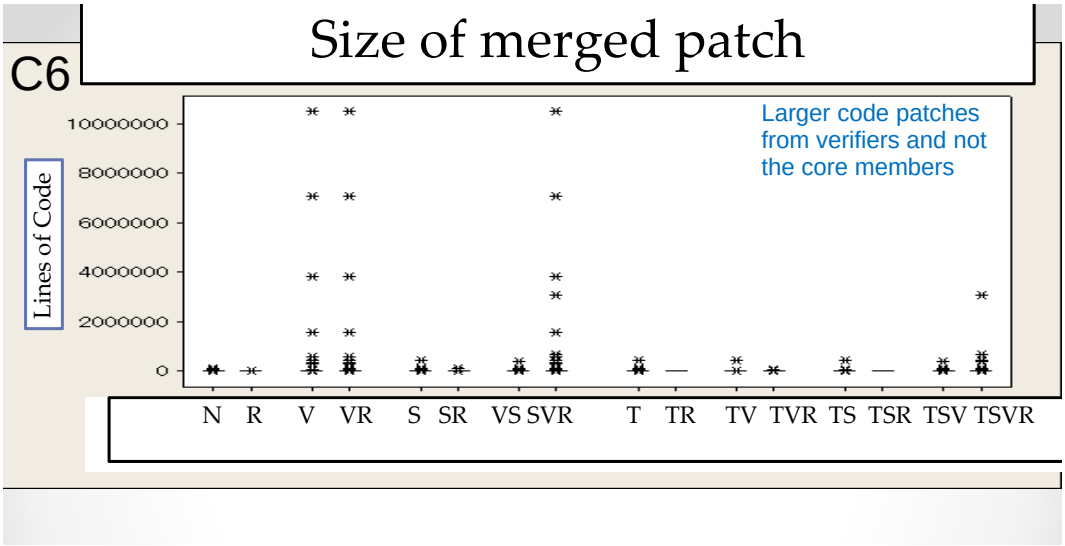


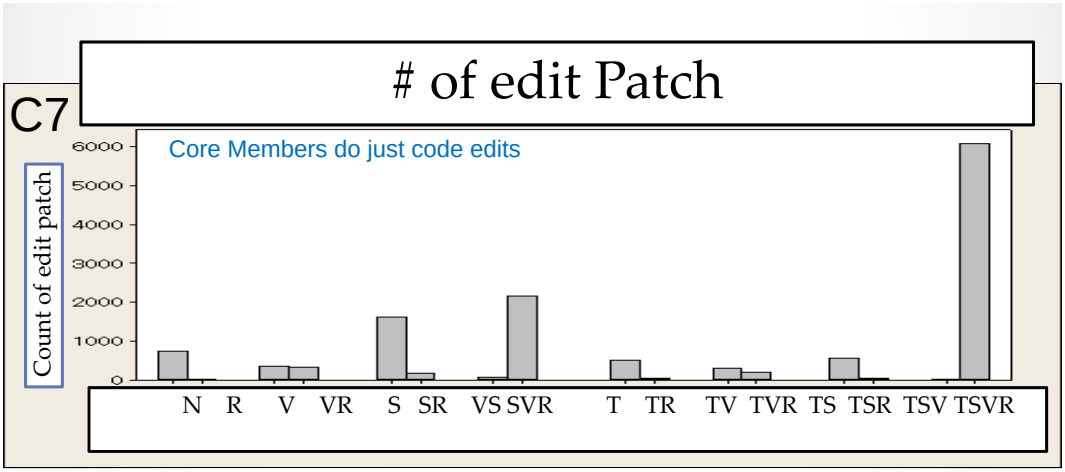


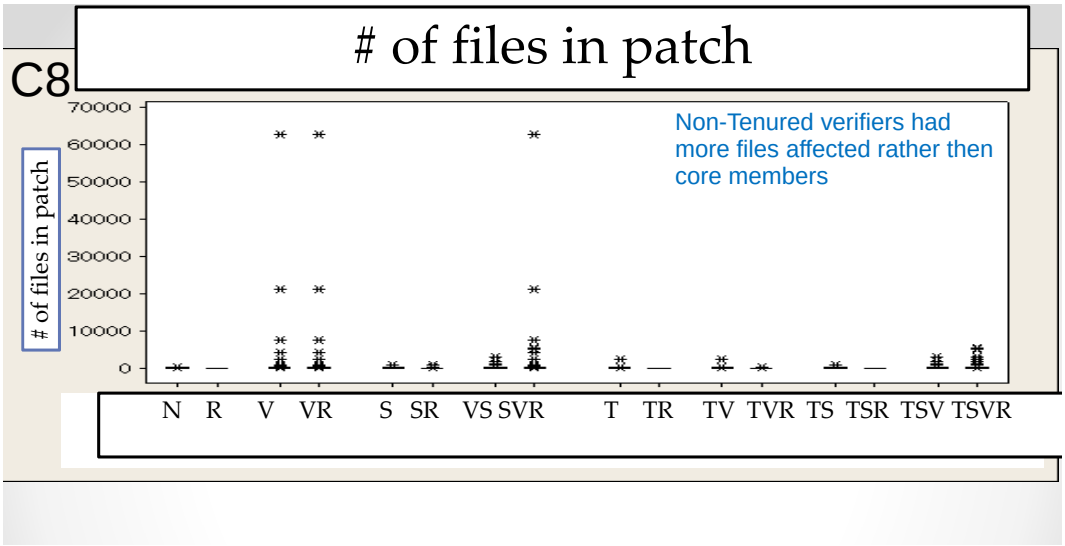

C3
# of Merged Patches
Core members have more merged patches
# of merged patches
3500
3000
2500
2000
1500
1000
500
0
N R V VR S SR VS SVR T TR TV TVR TS TSR TSV TSVR


C4
# of Abandoned Patches
Submitters and Core Members
Have more abandoned patches
# of abandoned patches
700
600
500
400
300
200
100
0
−100
N R V VR S SR VS SVR T TR TV TVR TS TSR TSV TSVR


C5
Size of deleted patch
No significance!
Lines of Code
1200000
1000000
800000
600000
400000
200000
0
N R V VR S SR VS SVR T TR TV TVR TS TSR TSV TSVR


C6
Size of merged patch
Larger code patches from verifiers and not the core members
Lines of Code
10000000
8000000
6000000
4000000
2000000
0
N R V VR S SR VS SVR T TR TV TVR TS TSR TSV TSVR


C7
# of edit Patch
Core Members do just code edits
Count of edit patch
6000
5000
4000
3000
2000
1000
0
N R V VR S SR VS SVR T TR TV TVR TS TSR TSV TSVR


C8
# of files in patch
Non-Tenured verifiers had more files affected rather then core members
# of files in patch
70000
60000
50000
40000
30000
20000
10000
0
N R V VR S SR VS SVR T TR TV TVR TS TSR TSV TSVR

References

[1] M. Aberdour. Achieving quality in open-source software. *IEEE Software*, 24(1):58 –64, jan.-feb. 2007.

[2] P. J. Adams, A. Capiluppi, and C. Boldyreff. Coordination and productivity issues in free software: The role of brooks' law. In *ICSM*, pages 319–328, 2009.

[3] B. Anda. Assessing software system maintainability using structural measures and expert assessments. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 204 –213, oct. 2007.

[4] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20:42–50, 2003.

[5] P. Anderson and M. Zarins. The codesurfer software understanding platform. *In Proc. IWPC'05*, pages 147–148, 2005.

[6] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE*, pages 361–370, 2006.

[7] A. April, J. H. Hayes, A. Abran, and R. R. Dumke. Software maintenance maturity model (smmm): the software maintenance process model. *Journal of Software Maintenance*, 17(3):197–223, 2005.

[8] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE*, pages 298–308, 2009.

[9] O. Armbrust, M. Katahira, Y. Miyamoto, J. Münch, H. Nakao, and A. Ocampo. Scoping software process models: initial concepts and experience from defining space standards. In *Proc. ICSP'08*, pages 160–172, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] L. J. Arthur. *Software evolution: The Software Maintenance Challenge*. Wiley-Interscience, New York, NY, USA, 1988.

[11] J. Asundi and R. Jayant. Patch review processes in open source software development communities: A comparative case study. In *Proc. HICSS '07*, pages 166c–, Washington, DC, USA, 2007. IEEE Computer Society.

[12] N. Baddoo and T. Hall. Motivators of software process improvement: an analysis of practitioners' views. *Journal of Systems and Software*, 62(2):85 – 96, 2002.

[13] N. Baddoo, T. Hall, and C. O'Keeffe. Using multi dimensional scaling to analyse software engineers' de-motivators for spi. *Software Process: Improvement and Practice*, 12(6):511–522, 2007.

[14] K. Balog and M. de Rijke. Determining expert profiles (with an application to expert finding). In *IJCAI*, pages 2657–2662, 2007.

[15] P. L. Bannerman. Macro-processes informing micro-processes: The case of software project performance. In *ICSP*, pages 12–23, 2008.

[16] V. R. Basili, G. Caldiera, and H. D. Rombach. *Experience Factory*. John Wiley and Sons, Inc., 2002.

[17] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[18] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful ... really? In *ICSM*, pages 337–345, 2008.

[19] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *ICSM*, pages 1–10, 2010.

[20] A. Bicego, P. Derks, P. Kuvaja, and D. Pfahl. Product focused process improvement: Experiences of applying the profes improvement methodology at dr 辰 ger. In *Proc. EuroMicro Conference*, 1999.

[21] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, April 2007.

[22] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In *Proc. ICSM'05*, pages 177 – 186, 2005.

[23] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *Proc. MSR'07*, pages 26–29, Washington, DC, USA, 2007.

[24] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? immigration in open source projects. In *Proc. MSR '07*, pages 6–14, Washington, DC, USA, 2007. IEEE Computer Society.

[25] B. Boehm and V. Basili. Top 10 list [software development]. *Computer*, 34(1):135 –137, jan 2001.

[26] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. *In Proc. MSR'06*, pages 105–111, 2006.

[27] A. Capiluppi, P. Lago, and M. Morisio. Characteristics of open source projects. In *Proc. CSMR'03*, pages 317 – 327, march 2003.

[28] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller. Open-source change logs. *Empirical Software Engineering*, 9:197–210, 2004.

[29] M. Christiansen and J. Johansen. *ImprovAbility*$^{\text{tm}}$ guidelines for low-maturity organizations. *Software Process: Improvement and Practice*, 13(4):319–325, 2008.

[30] P. Colla and J. Montagna. Framework to evaluate software process improvement in small organizations. In *Making Globally Distributed Software Development a Success Story, ICSP*, LNCS5007, pages 36–50. Springer Berlin, 2008.

[31] J. W. Creswell. *Research Design : Qualitative, Quantitative, and Mixed Methods Approaches*. Thousand Oaks, EUA : Sage, 2002.

[32] K. Crowston and J. Howison. The social structure of Open Source Software development teams. In *OASIS 2003 Workshop (IFIP 8.2 WG)*, 2003.

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

[34] K. E. Emam. *Spice: The Theory and Practice of Software Process Improvement and Capability Determination*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997.

[35] N. E. Fenton and M. Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, pages 149 – 157, 1999.

[36] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. *In Proc. WCRE'03*, page 90, 2003.

[37] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. ICSM'03*, pages 23–32, 2003.

[38] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *IEEE Trans. Soft. Eng.*, 17(8):751 –761, Aug 1991.

[39] D. M. Germán. An empirical study of fine-grained software modifications. In *ICSM*, pages 316–325, 2004.

[40] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior code changes. *Information and Software Technology*, 51(10):1394–1408, Oct. 2009.

[41] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 55–64, New York, NY, USA, 2010. ACM.

[42] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 495–504, New York, NY, USA, 2010. ACM.

[43] T. Hall, S. Beecham, and A. Rainer. Requirements problems in twelve software companies: an empirical analysis. In *Proc. IEE Software*, volume 149, pages 153 – 160, Oct. 2002.

[44] T. Hall and P. Wernick. Program slicing metrics and evolvability: an initial study. pages 35 – 40, sept. 2005.

[45] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Proc. ICSM'04*, pages 284 – 293, Sept. 2004.

[46] A. E. Hassan. Predicting faults using the complexity of code changes. *In Proc. ICSE'09*, pages 78–88, 2009.

[47] H. Hata. *Fault-prone Module Prediction Using Version Histories.* PhD thesis, Osaka University, 2012.

[48] J. D. Herbsleb and D. R. Goldenson. A systematic survey of cmm experience and results. In *Proc. ICSE '96*, pages 323–330, 1996.

[49] D. C. Hoaglin, F. Mosteller, and J. W. Tukey. Understanding robust and exploratory data anlysis. *Wiley Series in Probability and Mathematical Statistics New York*, 1983.

[50] J. Howison and K. Crowston. The perils and pitfalls of mining source-forge. In *Proc. MSR'04*, pages 7–11, Edinburgh, UK,, 2004.

[51] C. Jensen and W. Scacchi. Role migration and advancement processes in ossd projects: A comparative case study. In *ICSE*, pages 364–374, 2007.

[52] M. Jørgensen and D. I. K. Sjøberg. Impact of experience on maintenance skills. *Journal of Software Maintenance*, 14(2):123–146, 2002.

[53] H. H. Kagdi, M. Hammad, and J. I. Maletic. Who can help me with this source code change? In *ICSM*, pages 157–166, 2008.

[54] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proc. ICSM'10*, pages 1 –10, sept. 2010.

[55] S. H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley, 1995.

[56] N. Khurshid, P. L. Bannerman, and M. Staples. Overcoming the first hurdle: Why organizations do not adopt cmmi. In *ICSP*, pages 38–49, 2009.

[57] S. Kim, J. E. James Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Soft. Eng.*, pages 181–196, 2008.

[58] S. Kim and E. J. Whitehead. How long did it take to fix bugs? pages 173–174, New York, New York, USA, 2006. ACM Press.

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

[60] R. G. Kula, A. E. Camargo Cruz, N. Yoshida, F. K. Hamasaki, Kazuki and, X. Yang, and H. Iida. Using profiling metrics to categorise peer review types in the android project. In *Proc. ISSRE'12*, Dallas, TX, USA, 2012.

[61] M. K. Kulpa and K. A. Johnson. *Interpreting the Cmmi: A Process Improvement Approach*. CRC Press, Inc., Boca Raton, FL, USA, 2003.

[62] H.-J. Kung. Quantitative method to determine software maintenance life cycle. In *ICSM*, pages 232–241, 2004.

[63] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7:49–76, 2002.

[64] K. Lakhani and R. Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. In *Perspectives on Free and Open Source Software*. Cambridge, Mass: MIT Press, 2005.

[65] T. Lawrie and C. Gacek. Issues of dependability in open source software development. *SIGSOFT Softw. Eng. Notes*, 27(3):34–37, May 2002.

[66] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. In *Proc. METRICS'97*, pages 20 –32, nov 1997.

[67] N. G. Lester, F. G. Wilkie, D. McFall, and M. P. Ware. Investigating the role of cmmi with expanding company size for small- to medium-sized enterprises. *Journal of Software Maintenance*, 22(1):17–31, 2010.

[68] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. volume 30, pages 296–305, New York, NY, USA, Sept. 2005.

[69] M. Mantyla and C. Lassenius. What types of defects are really discovered in code reviews? *IEEE Trans. Soft. Eng.*, 35(3):430–448, May-June 2009.

[70] S. Massoni, M. Olteanu, and P. Rousset. Career-path analysis using optimal matching and self-organizing maps. In *Proc. WSOM '09*, pages 154–162, Berlin, Heidelberg, 2009. Springer-Verlag.

[71] F. McCaffery and G. Coleman. Lightweight spi assessments: what is the real cost? *Software Process: Improvement and Practice*, 14(5):271–278, 2009.

[72] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proc. CSCW '00*, CSCW '00, pages 231–240, NY, USA, 2000. ACM.

[73] T. Menzies, J. D. Stefano, C. Cunanan, and R. Chapman. Mining repositories to assist in project planning and resource allocation. *IEE Seminar Digests*, 2004(917):75–79, 2004.

[74] T. Meyers and D. Binkley. Slice-based cohesion metrics and software intervention. In *Proc. WCRE'04*, pages 256 – 265, nov. 2004.

[75] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 67–77, Washington, DC, USA, 2009. IEEE Computer Society.

[76] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[77] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proc. ICSE '02*, pages 503–512. ACM, 2002.

[78] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In *ICSE*, pages 274–284, 2003.

[79] S. Morisaki and H. Iida. Fine-grained software process analysis to ongoing distributed software development. In *Proc. SOFTPIT 2007*, volume 7, pages 26–30. Munich, Germany, 2007.

[80] J. C. Munson and S. G. Elbaum. Code churn: a measure for estimating the impact of code change. In *In Proc. ICSM'98*, pages 24–31, 1998.

[81] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. pages 364 –373, Sept. 2007.

[82] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324, New York, NY, USA, 2010. ACM.

[83] M. Niazi and M. Babar. De-motivators of software process improvement: An analysis of vietnamese practitioners views. In *Proc. PROFES 2007*, LNCS4589, pages 118–131. Springer Berlin / Heidelberg, 2007.

[84] M. Niazi, M. A. Babar, and N. M. Katugampola. Demotivators of software process improvement: an empirical investigation. *Software Process: Improvement and Practice*, 13(3):249–264, 2008.

[85] K. Nishizono, S. Morisakl, R. Vivanco, and K. ichi Matsumoto. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - an empirical study with industry practitioners. In *ICSM*, pages 473–481, 2011.

[86] B. Nuseibeh, A. Finkelstein, and J. Kramer. Fine-grain process modelling. In *IWSSD*, pages 42–46, 1993.

[87] L. Osterweil. Unifying microprocess and macroprocess research. In M. Li, B. Boehm, and L. Osterweil, editors, *Unifying the Software Process Spectrum*, volume 3840 of *Lecture Notes in Computer Science*, pages 68–74. Springer Berlin Heidelberg, 2006.

[88] K. Pan, S. Kim, and J. Whitehead, E.J. Bug classification using program slicing metrics. pages 31 –42, sept. 2006.

[89] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. SIGSOFT '08/FSE-16*, pages 226–237. ACM, 2008.

[90] F. J. Pino, J. A. H. Alegria, J. C. Vidal, F. García, and M. Piattini. A process for driving process improvement in vses. In *Trustworthy Software Development Processes, ICSP*, LNCS5543, pages 342–353. Springer Berlin, 2009.

[91] F. J. Pino, J. A. H. Alegria, J. C. Vidal, F. García, and M. Piattini. A process for driving process improvement in vses. In *ICSP*, pages 342–353, 2009.

[92] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[93] A. Rainer and T. Hall. Key success factors for implementing software process improvement: a maturity-based analysis. *Journal of Systems and Software*, 62(2):71 – 84, 2002.

[94] E. S. Raymond. *The Cathedral; the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly Media Inc., Sebastopol, 2008.

[95] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proc. ICSE'08*, pages 541–550, New York, NY, USA, 2008.

[96] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 541–550, New York, NY, USA, 2011. ACM.

[97] P. Rousset and J.-F. Giret. Classifying qualitative time series with som: The typology of career paths in france. In *Computational and Ambient Intelligence*, volume 4507 of *LNCS*, pages 757–764. Springer Berlin Heidelberg, 2007.

[98] M. Sanders and I. Richardson. Research into long-term improvements in small- to medium-sized organisations using spice as a framework for standards. *Software Process: Improvement and Practice*, 12(4):351–359, 2007.

[99] C. H. Schmauch. *ISO 9000 for Software Developers.* ASQ Quality Press, 2nd edition, 1995.

[100] B. D. Sethanandha. Improving open source software patch contribution process: methods and tools. In *Proc. ICSE '11*, pages 1134–1135, New York, NY, USA, 2011. ACM.

[101] J. Sillito and E. Wynn. The social context of software maintenance. In *ICSM*, pages 325–334, 2007.

[102] S. Sim and R. Holt. The ramp-up problem in software projects: a case study of how software immigrants naturalize. In *Proc. ICSE'98*, pages 361 –370, apr 1998.

[103] H. M. Sneed and P. Brössler. Critical success factors in software maintenance-a case study. In *ICSM*, pages 190–198, 2003.

[104] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[105] M. Staples, M. Niazi, R. Jeffery, A. Abrahams, P. Byatt, and R. Murphy. An exploratory study of why organizations do not adopt cmmi. *Journal of Systems and Software*, 80(6):883 – 895, 2007.

[106] P. S. Taylor, D. Greer, G. Coleman, K. McDaid, and F. Keenan. Preparing small software companies for tailored agile method adoption: Minimally intrusive risk assessment. *Software Process: Improvement and Practice*, 13(5):421–437, 2008.

[107] A. Tuffley, B. Grove, and G. McNair. Spice for small organisations. *Software Process: Improvement and Practice*, 9(1):23–31, 2004.

[108] C. G. von Wangenheim, T. Varkoi, and C. F. Salviano. Standard based software process assessments in small companies. *Software Process: Improvement and Practice*, 11(3):329–335, 2006.

[109] M. Weiser. Program slicing. In *Proc. ICSE'81*, pages 439–449, Piscataway, NJ, USA, 1981.

[110] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proc. MSR'07*, pages 1–10, 2007.

[111] F. G. Wilkie, D. McFall, and F. McCaffery. An evaluation of cmmi process areas for small- to medium-sized software development organisations. *Software Process: Improvement and Practice*, 10(2):189–201, 2005.

[112] Y. Ye and K. Kishida. Toward an understanding of the motivation of open source software developers. In *ICSE*, pages 419–429, 2003.

[113] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Soft. Eng.*, 30(9):574 – 586, Sept. 2004.

[114] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, Sept. 2004.

[115] S. Yip and T. Lam. A software maintenance survey. In *Proc APEC'94*, pages 70 –79, Dec 1994.

[116] C. Yoo, J. Yoon, B. Lee, C. Lee, J. Lee, S. Hyun, and C. Wu. A unified model for the implementation of both iso 9001:2000 and cmmi by iso-certified organizations. *Journal of Systems and Software*, 79(7):954 – 961, 2006.

[117] L. Yu. Indirectly predicting the maintenance effort of open-source software. *Journal of Software Maintenance*, 18(5):311–332, 2006.

[118] L. Yu. Mining change logs and release notes to understand software maintenance and evolution. *CLEI Electronic Journal*, 12(2):1 – 10, 2009.

[119] S. Zhang, Y. Wang, Y. Yang, and J. Xiao. Capability assessment of individual software development processes using software repositories and dea. In *ICSP*, pages 147–159, 2008.

[120] L. Zhu, D. R. Jeffery, M. Staples, M. Huo, and T. T. Tran. Effects of architecture and technical development process on micro-process. In *ICSP*, pages 49–60, 2007.

[121] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. *In Proc. ICSE'08*, pages 531–540, 2008.