FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Design and Evaluation of a Natural Language Processing Based Methodology for Classification and Profiling of Artifacts in Software Evolution

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Sozial- und Wirtschaftswissenschaften

eingereicht von

Mag. Andreas Mauczka
Matrikelnummer 0125851

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Thomas Grechenig

Diese Dissertation haben begutachtet:

(Prof. Dr. Thomas Grechenig) (Prof. Dr. Rudolf Freund)

Wien, 25.08.2016

(Mag. Andreas Mauczka)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Design and Evaluation of a Natural Language Processing Based Methodology for Classification and Profiling of Artifacts in Software Evolution

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Sozial- und Wirtschaftswissenschaften

by

Mag. Andreas Mauczka
Registration Number 0125851

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Thomas Grechenig

The dissertation has been reviewed by:

(Prof. Dr. Thomas Grechenig) (Prof. Dr. Rudolf Freund)

Wien, 25.08.2016

(Mag. Andreas Mauczka)

Erklärung zur Verfassung der Arbeit

Mag. Andreas Mauczka
Mollardgasse 22/2/30, 1060 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_____ _____

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Ich möchte mich in erster Linie bei Prof. Thomas Grechenig für seine Unterstützung und motivatorischen Künste bedanken, der mir in entscheidenden Phasen meines Forscherlebens Mut zugesprochen hat.

Desweiteren möchte ich mich bei Christian Schanes bedanken, der mich durch seine geduldigen und detaillierten Anmerkungen in die richtige Richtung gewiesen hat, sowie bei Thomas Moser, der ebenfalls mit seinen Hinweisen und Hilfestellungen zur Qualität der Arbeit beigetragen hat.

Florian Brosch, Thomas Wagner und Markus Huber, die wesentlichen Anteil an der praktischen Durchführung der Experimente und der Implementierung der Werkzeuge hatten, gebührt ebenfalls Dank.

Ich widme diese Arbeit Sonja, Luisa und Maria.

Kurzfassung

Zahlreiche Methoden und Ansätze wurden im Laufe der Jahre entwickelt, um Erkenntnisse über die Personen und Prozesse zu gewinnen, die die Evolution von Software weiter treiben. Aktuelle Studien setzen sich unter anderem mit dem Einsatz von Natural Language Processing (NLP) in diesem Bereich auseinander, um damit Informationen aus verfügbaren Repositories (z.B. aus Bug Tracking-Systemen oder Version Control-Systemen) zu gewinnen.

Im Rahmen der vorliegenden Arbeit wird die SubCat-Methode vorgestellt, ein Ansatz, der Projektmanagern und Forschern NLP und Data Mining Funktionalität im Rahmen eines Frameworks zur Verfügung stellt. Die Methode wurde ausgehend von existierenden Problemstellungen und dem State-of-the-Art in der Erforschung von Software Evolution entworfen.

Der vorgestellte Ansatz wurde in mehreren Szenarien eingesetzt, um die Effizienz der Methode zu validieren. Dabei konnten unterschiedliche Aspekte von Software Evolution beleuchtet und neue Erkenntnisse gefunden sowie existierende Annahmen teilweise widerlegt werden. Die Methode wurde in einem breiten Kontext eingesetzt, der von der Klassifizierung von Modulen und Code Changes bis zur Sentiment Analysis von Kommentaren in Bug Trackern reicht. Unter anderem wurde die Methode verwendet um sicherheitsrelevante Änderungen zu identifizieren und mittels Security Advisories zu validieren. Des weiteren wurde die Methode genutzt um Reports für einen Bug Tracker aus einem Code Repository zu generieren. Dies stellt einen potentiellen Nutzen für Projekte dar, die erst zu einem späteren Zeitpunkt einen Bug Tracker einsetzen.

Zusätzlich zu den erwähnten Szenarien wurde ein Klassifizierungsmechanismus für Code Changes in Wartungskategorien entwickelt und validiert. Dieser wurde gemeinsam mit Sentiment Analyse eingesetzt, um eine mögliche Nutzung zur Profilierung von Befinden und Tätigkeiten von Entwicklern zu illustrieren. Diese Profile können in einem nächsten Schritt für die Analyse von Langzeitmotivatoren in Projekten genutzt werden oder auch im Rahmen von Dashboards für Projektmanager Auskunft über aktuelle Schwerpunkte geben.

Schlagwörter: *Mining Software Repositories, Natural Language Processing, Software Evolution, Software Maintenance*

Abstract

Software evolution is an active field of research and has featured many different approaches to learn more about the processes and people that drive software engineering efforts. Recent studies have further advanced research on software evolution by incorporating Natural Language Processing methodologies to mine textual artifacts accessible in repositories like Bug Tracking Systems and Version Control Systems for information about the nature of software engineering.

We propose a methodology called SubCat that exploits Natural Language Processing and data mining capabilities to provide a framework that provides both researchers and managers access to software evolution meta-data contained within their repositories. The proposed methodology incorporates in its design the current state of the art in the mining of software repositories and answers defined problems with current tool support.

We apply the resulting framework in different scenarios to validate the methods efficiency. In these scenarios various aspects of software evolution were analyzed, new findings could be made and existing assumptions partially refuted. The methodology was applied to cover a broad range of topics from classification of code changes to using Sentiment Analysis on comments in a bug tracker. Further, the methodology was used to identify security-relevant changes, which could be validated by using existing Security Advisories. Additionally, we employ the framework to generate content for a Bug Tracker based on information available in a Code Repository to showcase a potential use for projects that did not start out with a Bug Tracker.

Aside the mentioned scenarios, we created a classification mechanism for code changes into maintenance categories and evaluated it for cross-project validity. The dictionary and the provided Sentiment Analysis capabilities of the framework were then used to generate developer profiles to showcase a potential use for future studies on longterm developer motivation or dashboards for project managers to see possible conflicts and problems at a glance.

Keywords: *Mining Software Repositories, Natural Language Processing, Software Evolution, Software Maintenance*

Contents

List of Figures

List of Tables

Listings

Part I

Introduction and Theory

CHAPTER 1

Introduction

Contents

1.1 General

Software maintenance is an expensive and time consuming task that concerns the whole software industry. In the 70ies first reports indicated that back then, already 40% of the effort spent in the software industry in Great Britain was invested into maintenance (Boehm(1973) cited by E.B. Swanson in [12]). In the 80ies this issue started to be addressed by researchers and a "maintenance problem" was defined, yet a consistent increase was reported by Zvegintzov in [13] and 50% of all effort were still spent in maintenance. The reason was that maintenance was an afterthought to development, so software was not designed for maintenance. The challenges of software maintenance back then were lack of robustness, bug injections and regressions caused by applying a change and the mind-set that maintenance was a post-delivery activity [14]. By the end of the 90ies the effort spent on maintenance even further increased to 85-90% [15] - existing software grew older and even harder to maintain and new software had to be maintained as well,

with the software industry constantly growing. As with software test, software maintenance came as an afterthought and even though models and approaches to create maintainable software existed since the 80ies, the industry up to this day is slow to pick up on these. However, since the maintenance problem is well-known, a lot of research has been undertaken in the formalization of the maintenance or software evolution process.

A software maintenance process in an industrial sense and the process of software evolution in the sense of open source projects are often used synonymously. This used to be problematic in the past, as research has shown that a software maintenance process in an industrial sense rarely existed in open source projects. Open source software evolves in a sense that it is continuously developed and ideally deployed. This evolution begins most of the times from an already existing project scaffolding (see Raymond's fundamental essay "The cathedral and the bazaar" [16]). Hence, the early, waterfall-like understanding of industrial software development with a decoupled software maintenance phase from industrial case studies from the '80ies simply did not apply to open source projects. Therefore, when we talk about software development processes, we talk about feature-wise software evolution (compare to the definition of *software evolution* in [2] by Bennet and Rajlich).

However, while the processes themselves used to be fundamentally different in open source and closed source development (see figure 1.1), state of the art software development processes like SCRUM (introduced by Sutherland and Schwaber in [17]) incorporate findings and best-practices of software evolution in open source projects. In fact, the iterative-incremental development process that uses small feature releases and constant delivery is noted by Raymond in 1999 in [16] already. However, while processes for pre- and post-delivery may have differed greatly between industrial and open source settings in the past, the performed tasks that deliver features already were quite similar (see Mockus [18], Hassan [19] and Mauczka et al. in [11]). Research findings in software evolution of open source projects may well be transferable to the closed source domain. This notion is supported by the fact that open source developers mostly develop industrial software for a living and the advent of agile development in standard industry best practices may well be accounted in some degree to open source practitioners applying lessons learned from software evolution with the latest *State of agile development*[1] survey showing that 95% of the participants work in organizations that practice agile.

As indicated earlier, to approach a formalization of a maintenance process and to identify the main driving forces in the maintenance and evolution phase, it is mandatory to analyze the currently existing systems that are in the maintenance and evolution phase of their life-cycle. One aspect of this analysis is in the extraction of information by means of data mining the

[1] `https://versionone.com/pdf/VersionOne-10th-Annual-State-of-Agile-Report.pdf`

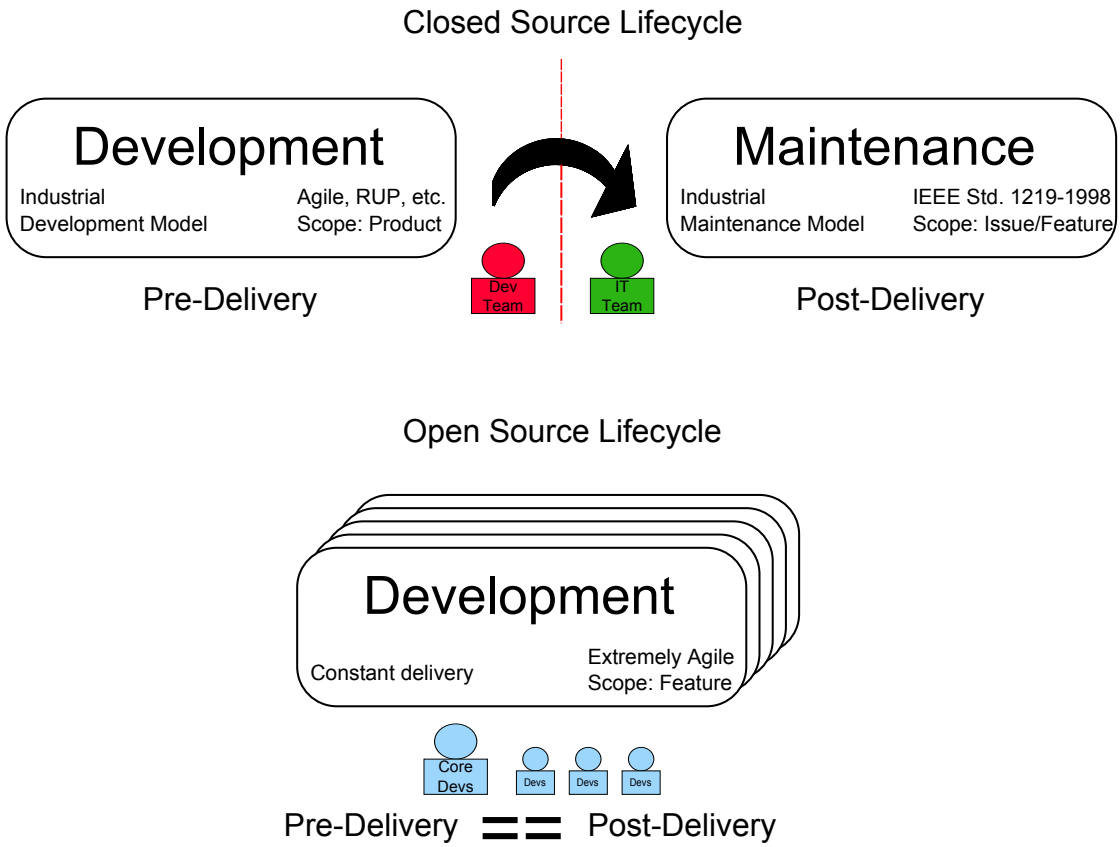


Figure 1.1:
Historical Overview of Open vs. Closed Software Lifecycle

accessible data sources of these existing systems. With the advent of software repositories like Concurrent Versions System[2] (CVS), Subversion[3] (SVN), GIT[4], etc. huge pools of information on software evolution and maintenance have become accessible to researchers. This opened up a whole new area of research on the maintenance process and allowed researchers to analyze very specific details on maintenance, see for e.g. [20] for research on bug injection during changes, [21] for security related analysis or [22] for coherent change detection in CVS. However, the increasing activity of the Open Source Movement of the late 90ies and Raymonds fundamental essay [16] ”The cathedral and the bazaar” together with the forming of the open source initiative had an arguably even bigger impact on software evolution research than software repositories had. Due to the nature of open source projects, all aspects of software evolution are largely available and comprehensive in most cases (e.g. software repositories, mailing lists, forums,

[2] http://www.nongnu.org/cvs/
[3] https://subversion.apache.org/
[4] https://git-scm.com/

issue trackers, etc.). Several studies and communities leverage these sources to learn new facets about software evolution and maintenance, e.g. see Rigby et al. in [23] for an analysis of mailing lists in Open Source Projects, or Hassan in [24] for leveraging code change complexity for fault prediction. Fischer et al. in [25] use software repositories and bug tracking systems to generate a release history database.

Latest research on software evolution centers around leveraging natural language processing (NLP) to unveil insights about maintenance artifacts. Studies in this area cover a large span of software artifacts. In [26] Abebe et al. analyze the connection between comment vocabulary and code vocabulary, while Fluri et al. in [27] analyze how code and comments evolve over time. La Becca in [28] uses textual descriptions during evolution for fault prediction, Hindle et al. in [29] use NLP techniques on source code to provide code completion. Sentiment mining (e.g. Murgia et al. [30], Guzman et al. [31]), which is also based on NLP, aims for insights on how open source project teams interact with each other, since social interaction is a major driving force behind long term contribution.

1.2 Motivation

Maintenance is a difficult and costly task and one that is impossible to avoid for any successful software project. It is therefore important to know how to deal with software maintenance and how to manage it properly. It is important to find out what drives the people doing maintenance work and how to keep them motivated. It is important to know which areas of a system will change and need to be changed. Is the tension in projects measurable in the language used before releases? Are there specialized roles in a project, which might be antagonistic to shared code ownership? Do developer task profiles change over time? Is there an expert profile and an initiative profile of maintenance tasks? Is it possible to identify security-relevant components of a software, just by analyzing the commit messages?

There are many more questions to be addressed in this field of research and software evolution has a long history indeed and is a well defined area. Models for maintenance have been developed as far back as the 80ies and the main problems have been sufficiently formally addressed. With the surface of large-scale open source projects, which keep constantly evolving and are never really finished, monitoring of maintenance activities is very important. There have been several attempts to help support software evolution by offering tools like ArchView in [32], EPOSee in [33], GlueTheos in [34], APFEL in [35], ROSE in [36], but there has also been a lot of fundamental research work to analyze the structure and processes of software evolution in open source projects (see section 2.1). However, all these approaches, be it exploratory research work or tool support, share the commonality that they are difficult to apply or the approach itself

lacks cross-project validity and cannot be used outside the testing set. This mainly stems from the fact that approaches such as Fischer et al. in [25] use many input sources that greatly vary between projects. For this work therefore the biggest challenge is to keep things simple and light-weight and to use an approach as direct as possible. This is reflected in the conception and the theoretical approach by focusing on natural language, but also in the architecture of the implementation.

We assume that meta-information contained within software development tools in natural language reflects the intentions of developers. We propose, based on findings in personal studies and in the current state of the art, that by processing this meta-information by using existing NLP tools like WordNet [5] more information about software evolution may be found than by applying code based analysis. In large parts of both industrial and open source projects, meaningful meta-information in software development tools is consistently available and thus enables us to further propose a light-weight approach to analyze this information.

Finally, we want to stress the intended simplicity of our methodology. In 2013, at the 10th anniversary of the Mining Software Repositories conference[6], Hemmati et al. analyzed over 100 full research papers that were published by the conference. One of their central lessons learned is the following, which we take to heart for our light-weight, simplified approach to software evolution analysis:

Simple analysis often outperform their complex counterparts [6]

1.3 Approach

The major goal of this thesis is to provide a methodology to analyze diverse sources of information about software projects with the focus on transportability and applicability of the approach. The presented approach in this thesis is therefore a light-weight approach and this notion is carried out consequently. We use English natural language artifacts to learn the intent of actions during software maintenance, since we believe that written statements are more reliable than interpreted source code. Prior work and the results in this thesis carry this notion. However, the extent to which NLP techniques are used is minimal since a complex set of rules infers with the applicability of the approach. Whenever possible, the simplest algorithm or heuristic is applied (e.g. the training algorithm for the dictionary presented in chapter 6).

The methodology also aims to integrate as many sources of information as is feasible under the consideration of usability. Accordingly, we strongly modularize the architecture of the

[5] http://wordnet.princeton.edu/

[6] msrconf.org

implemented tooling to benefit from the strength of our generic approach. Due to our simplistic approach, the SubCat methodology is applicable for any programming language and various types of repositories. Furthermore, the entity model that is at the base of SubCat is built around these considerations of integration of various repository types. In the current implementation, GIT and SVN as Version Control System (VCS) and Bugzilla as Bug Tracking System (BTS) are supported, but any of the two repository types may be analyzed if corresponding miner modules are added, since a robust model is in place that allows easy customization to include e.g. JIRA or Mercurial. The SubCat methodology encompasses existing and customized NLP approaches, e.g. one supported mechanism is based on prior work of Mockus in [37] and Hassan in [19].

1.4 Contributions

The principal idea of the SubCat methodology is to offer researchers and managers a fast overview about the current state of a project to help researchers develop new theories and insights and to provide project managers or community moderators with self-explanatory charts and reports to learn more about their community (motivation) and project (technical metrics). The methodology is light-weight and easily extendable; furthermore it is easy to setup and easy to use. The implementation of the methodology visualizes the information extracted in a meaningful way to the end-user. It also may be easily expanded and customized.

This thesis therefore contributes the following items to the research of software evolution:

1. A methodology to gather information and provide insights into software evolution processes

    a) The methodology is based on existing and implemented NLP methods and components (e.g. sentiment analysis and the Stanford NLP library[7])

    b) The methodology uses resources available in most open-source and industrial software projects (i.e. issue trackers and code repositories)

    c) The methodology is transferable between programming languages (e.g. it may be applied to both java and C++) and domains (i.e. it may be used to analyze projects in the open source as well the closed source domain)

    d) The methodology is capable of providing metrics about both a software product (e.g. identify problematic software components) and a software project (e.g. identify project member roles)

[7] `http://nlp.stanford.edu/software/`

e) The methodology supports customization of analysis methodologies and provides integration points for future methods and components

2. An implementation of the methodology as a java-based tool with the following core features

   a) Data extraction and normalization into a database for VCS and BTS

   b) Implementation of a dictionary-based approach for classification of artifacts mined in BTS and VCS

   c) Sentiment analysis capability for text bodies in VCS and BTS to show morale and team motivation over time

   d) Convenient mining tools like account matching or identification of technical information in bodies of text

3. And finally, the application of the methodology to:

   a) Create a cross-project valid dictionary to classify commit messages into maintenance categories

   b) Identify security relevant changes during software evolution

   c) Populate a bug tracker by using corrective changes discovered by an implementation of SubCat

   d) Identify developer profiles in open source projects based on their actions and sentiments expressed in messages and comments

   e) Provide a survey on change classification by the original authors of changes to the source code

1.5 Structure of the Thesis

This thesis is structured as follows:

- **Part I - Introduction and Theory**
  - **Introduction:** a short introduction into the motivation and goals of the thesis at hand
  - **State of the Art:** this chapter covers the fundamentals and theoretical groundwork on software maintenance in industrial settings and the open source domain as well as existing approaches in the research area of software evolution to leverage data mined from different types of repositories

- **Part II - Design**
  - **Designing a Framework to Use NLP Techniques for Data Mining in Software Repositories:** the chapter describes the idea and the concepts behind the SubCat methodology, as well as the implied architectural design and the entity model to reflect the proposed methodology
  - **Implementation of the SubCat Methodology:** in this chapter the current implementation of core functionality of the SubCat methodology is described
- **Part III - Application**
  - **Applying the SubCat Methodology for a Preleminary Feasability Study:** a preleminary feasibility study was conducted to see whether the proposed methodology could be implemented
  - **Applying the SubCat Methodology for Change Classification:** to showcase the applicability of the methodology, we create a cross-project dictionary, which is used by an implementation of SubCat to classify commit messages into Swanson's maintenance categories
  - **Applying the SubCat Methodology for Security Analysis:** since the SubCat methodology is generic, this study shows how an implementation of SubCat may be used to identify security critical components and changes
  - **Applying the SubCat Methodology to Populate an Issue Tracker:** a practical application of an implementation of SubCat deals with the re-engineering of a BTS on the basis of commit messages from a VCS
  - **Applying the SubCat Methodology to Create Developer Profiles:** SubCat is configured to mine exemplary profiles based on categorized tasks in VCS and sentiment and activity in BTS
  - **Applying the SubCat Methodology in a Survey for Commit Classification:** we use the pre-processing capabilities of the current SubCat implementation to provide data for a ground-truth survey on change classification
- **Part IV - Outcome**
  - **Conclusion**: this chapter holds an overview of the contributions of this thesis and the applications of the presented methodology
  - **Future Works**: this chapter describes the possible next steps for the methodology and the implementation itself and discusses new appliances of SubCat

1.6 List of Publications

The work in this thesis was presented at academic and peer-reviewed conferences, workshops and published as technical reports.

1. A. Mauczka, M. Bernhart and T. Grechenig. Analyzing the Relationship of Process Metrics And Classified Changes - A Pilot Study. In *SEKE 2010*, pages 269-272, 2010. [10]

2. A. Mauczka, C. Schanes, F. Fankhauser, M. Bernhart and T. Grechenig. Mining Security Changes in FreeBSD. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 90-93, 2010. [38]

3. A. Mauczka, M. Huber, C. Schanes, W. Schramm, M. Bernhart and T. Grechenig. Tracing Your Maintenance Work–A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages. In *Fundamental Approaches to Software Engineering*, pages 301-315, 2012. [11]

4. A. Mauczka, F. Brosch, C. Schanes, and T. Grechenig. Dataset of Developer-Labeled Commit Messages. In *12th Working Conference on Mining Software Repositories (MSR)*, 2015. [39]

5. F. Brosch, A. Mauczka and T. Grechenig. Technical Report: Implementing a light-weight tool in the MSR area by applying NLP methods, *Technical Report, 2015*. [40]

6. M. Bernhart, A. Mauczka, M. Fiedler, S. Strobl and T. Grechenig. Incremental reengineering and migration of a 40 year old airport operations system. In *IEEE International Conference on Software Maintenance*, pages 503-510, 2012. [4]

7. T.S. Wagner and A. Mauczka. Technical Report: Populating a bug database by using repositories. *Technical Report*, 2015. [41]

CHAPTER 2

State of the Art

Contents

There is no such thing as a new idea.

Mark Twain in [42]

The following chapter provides an overview of the field of software evolution and maintenance and shows how practices converged in the last years. It covers the fundamental work of research that has been published on data mining in software evolution research and of its impact on the thesis at hand. This chapter is therefore structured as follows:

- Section **Software evolution and maintenance** provides historical insight into software maintenance in industrial projects
- Section **Commonalities of Software Development and Software Evolution** explains the nature of open source projects and how and why the open source development process used to differ to an industrial project setup and converged over the last years

- Section **Case Study: Building Maintainable Software** features a case study on agile software development practices and software maintenance parallels
- Section **Organizational and Social Aspects of Software Evolution** describes software evolution from a participants point of view
- Section **Mining Software Repositories** explains the current state of the art in software evolution and maintenance research

2.1 Software Evolution and Maintenance

A lot of resources in software development have been dedicated to tackling the "maintenance problem" that was identified in the 80ies. These days, every developer is confronted with the issue of maintaining software, as almost no piece of software exists solely by itself. Even in a greenfield approach, one reuses libraries or functionality, which constantly evolve and thus need to be maintained. Development methods like SCRUM (introduced by Ken Schwaber during [**?**]) or XP [43] preach of constant change of a system to ease future maintenance of software. Development frameworks like Spring[1] provide patterns and strategies to enforce maintainable code. Continuous integration and test automation efforts also provide means to an easier software maintenance phase. However, some of these frameworks and methods also add dangers to software maintenance - undisciplined use of agile methods will lead to unmaintainable code. High learning curves on development frameworks lead to developers being overwhelmed by complexity and the improper use of frameworks. Additionally, maintenance of external frameworks also becomes an issue. High degrees of test automation lead to maintenance problems in the test code written for these testing frameworks. These issues are largely of a technical nature, however, there are also soft factors to consider, like developer motivation and psychological factors of maintenance work, which have been identified as early as in the first studies by Swanson [12].

As the "maintenance problem" is constantly growing along with the software industry, it is being addressed by software architects and developers, but also managers, who have grown sensitive to the issue. The categorical thinking in pre-delivery and post-delivery activties is starting to diminish by techniques like continuous integration and processes like XP or SCRUM and shows the convergence of open source development organization and industrial software development and maintenance (see e.g. Stefan Koch in [44] on the similarities of open source development and agile practices).

[1] `http://spring.io/`

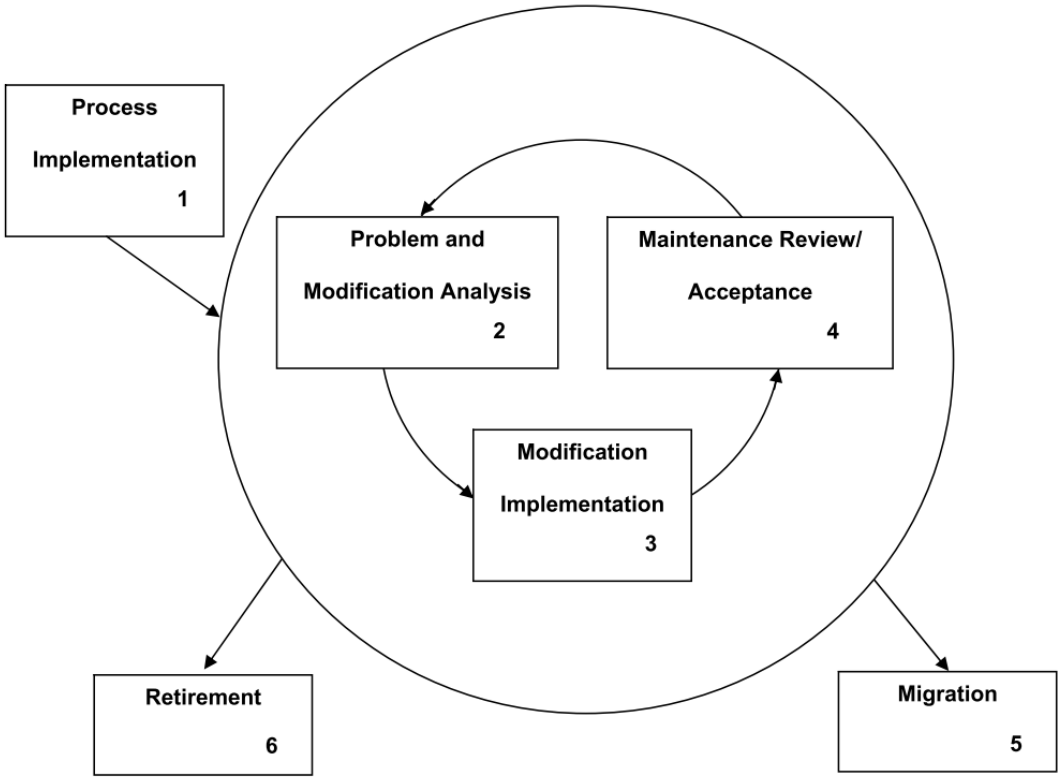


Figure 2.1:
The Maintenance Process as Defined in IEEE Standard 14764-2006 [1]

The Industrial Software Maintenance Process

In this section several software maintenance processes are described and compared. As a common starting ground, we present the IEEE Standard 14764-2006 for a definition of a maintenance process and terms relevant in software maintenance. The standard describes software maintenance from the point of view as a service provided after software development is finished, which is arguably outdated with the advent of agile methodologies.

IEEE Standard 14764-2006 Software Engineering - Software Life Cycle Processes - Maintenance

Before we discuss historical and actual maintenance processes and best practices in current literature, it is important to call into memory the IEEE Standard 14764-2006 [1] that covers a standardized maintenance process.

As can be seen in figure 2.1, the process consists of five steps:

1. **Process Implementation:** This step concerns itself with the time before actual maintenance work is done. It describes necessary tasks and activities that are required in order to plan and prepare for maintenance and the transition from development to maintenance

2. **Problem and Modification Analysis:** This is the first actual maintenance activity - during this step a problem or possible modification is analyzed, replicated, classified, etc.

3. **Modification Implementation:** In this step the implementation of the potential fix or enhancement, as well as quality assurance steps like testing and updating the system documentation, are done.

4. **Maintenance Review/Acceptance:** During this step the software is, similar to the regular development life cycle, reviewed and/or accepted. This might occur in any agreed upon setting, e.g. it might be a formal audit, or a user acceptance test

5. **Migration:** This is a very specialized maintenance process step. During this step, the system is migrated to a new environment. It is arguable that this step is part of a normal maintenance process, since depending on the complexity of the system and the migration plan, a migration project might actually be a development project in itself - see, e.g. [11] or [45] for migration projects

6. **Retirement:** This is the last step of the maintenance process and it describes the tasks and activities that are required to retire a piece of software

In IEEE Standard 14764-2006 [1] there are also relevant definitions for terms commonly used for maintenance items and activities. These terms are broadly used in maintenance literature. Firstly, different types of maintenance are defined in [1] as:

> "**adaptive maintenance** [is] the modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment"
>
> "**corrective maintenance** [is] the reactive modification of a software product performed after delivery to correct discovered problems"
>
> "**emergency maintenance** [is] an unscheduled modification performed to temporarily keep a system operational pending corrective maintenance"
>
> "**perfective maintenance** [is] the modification of a software product after delivery to detect and correct latent faults in the software product before they are manifested as failures"

"**preventive maintenance** [is] the modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults"

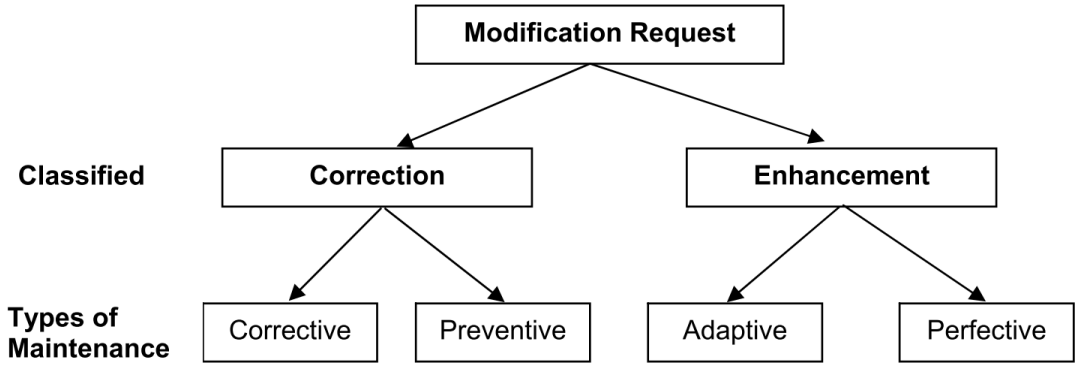


Figure 2.2:
Classification of Modification Requests in IEEE Standard 14764-2006 [1]

Secondly, the term of Modification Request (MR) is defined as "[..] a generic term used to identify proposed modifications to a software product that is being maintained" ([1]). An MR may be classified into types of maintenance, as can be seen in figure 2.2

Related Works on Industrial Software Maintenance Processes

Software maintenance as a problematic field in the software lifecycle has been identified as early as the 70ies, when already 40% of the effort spent in the software industry in Great Britain was being invested into maintenance (Boehm(1973) cited by E.B. Swanson in [12]). The problem was referred to as "an iceberg" by Swanson in "The dimensions of maintenance" [12]. The main issue therefore was lack of a formalized approach to measuring of software maintenance. In an effort to formalize the field of software maintenance, Swanson defined different types of maintenance tasks and measures that should be applied to maintenance tasks and processes. Swanson defined corrective, adaptive and perfective maintenance (compare to IEEE Standard 14764-2006 [1]). In 1974, Lehman first formulated his famous laws on software evolution, which he revised 1980 [46] and stated that in the US "Of the total U.S. expenditure some for 1977, 70% was spent on programm *maintenance* and only 30% on program *development*". He immediately puts this figure into perspective by stating that maintenance in this context refers to any change to the software after its first installation, still, considering the situation at the

beginning of the 70ies, an alarming increase in software maintenance could be seen. Lehman deducted his famous laws from his findings and later revised them again in 1997 [47]:

Continuing Change A system will change until it is more cost effective to replace the system

Increasing Complexity A system grows more complex over time if its structure (a modern take on this is presented by Fowler in his essay "Is Design dead?" - Lehman's structure is essentially software design) is not actively being maintained

Self Regulation Evolution of systems is self regulating within certain parameters

Conservation of Organizational Stability (Invariant Work Rate) Lehman claims that global activity rate in a project is statistically invariant over product lifetime

Conservation of Familiarity (Perceived Complexity) Lehman proposes that in order for the stakeholders of a system to keep up with system change, system growth is restricted and invariant as the system evolves

Continuing growth A system must grow in order to keep up with user expectations and satisfaction

Declining Quality This law states that quality of systems will decline unless they are rigorously maintained and constantly adapted

Feedback System The system is required to include mechanisms to allow monitoring of the system and give feedback

These laws were a great step to understand the problematic field of software maintenance. In 1981, Lientz et al. [48] presented a survey based on prior work together with Swanson, which allowed a more fine-grained view into the ongoings of industrial software maintenance. They found that "[..] departments tend to spend half of their application staff time on maintenance [..] [that] over 40% of the effort [..] is spent on providing user enhancements and extensions". The main findings of their survey were that the relationship between software user and maintenance provider is critical for a products success. They directly relate lack of user understanding and inadequate training to problems in maintenance. They also identified user demands as the largest source of system dissatisfaction during maintenance. They suggest that during maintenance the users should be put more into focus.

In 2000, Bennett and Rajlich published a road-map for software maintenance and evolution in [2]. As Swanson mentioned already back in 1976, they too point out that "[..] much

more empirical knowledge about software maintenance and evolution is needed, including process, organization and human aspects". Bennett and Rajlich proposed that the current view on maintenance was too simplistic as it focused only on the post delivery activities and pointed out that tasks during maintenance were diverse and not restricted to post-delivery. They therefore present a staged model for the software life cycle. They identify the following stages (see also figure 2.3):

1. **Initial development:** this is the initial version of the system
2. **Evolution:** this stage aims at adapting the application to user needs and environment changes. Corrective measures also take place here
3. **Servicing:** during this stage, only minor changes to the system are feasible. Either architectural or internal team knowledge have been lost or decayed and the software may no longer evolve
4. **Phase-out:** the system is no longer being maintained, but may still be in use
5. **Close-down:** the system is being shut down and users are directed toward other systems

The profile of maintenance activities depends therefore on the life-cycle stage that the system is currently in.

In their survey paper of 2008 [49], Godfrey and German make an update to adaptive, perfective and corrective maintenance tasks, proposing that:

Corrective maintenance are changes that fix bugs in the codebase

Adaptive maintenance are changes that allow a system to run within a new technical infrastructure

Perfective maintenance are any other enhancements intended to make the system better, such as adding new features, boosting performance, or improving system documentation

This definition is problematic in so far as e.g. a change that allows a system to run within a new technical infrastructure might be synonymous with a boost in performance. Therefore one task is only differentiable by intent, but not by result. This is important as for an automated approach of classification, an unambiguous definition of maintenance tasks is necessary, with as little overlap as possible. Godfrey and German also discuss the simple staged model for software evolution and stress its importance as a descriptive model for software evolution, as it shows that

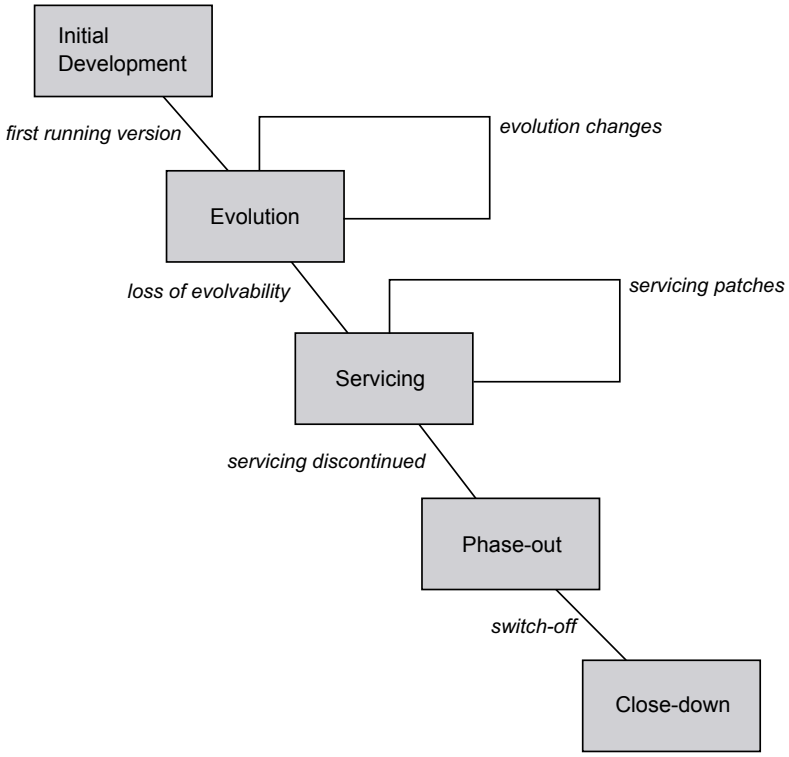


Figure 2.3:
Simple Staged Model by Bennett and Rajlich [2]

software development and maintenance as well as evolution over-lap, that software development and maintenance/evolution blur in incremental and iterative software development models.

Godfrey and German further discuss Lehman's laws of evolution and point out that open source projects like the Linux kernel violate a number of these laws. They argue that open source largely differs form industrial projects. Thus, in the next chapter, we will discuss the topic of software maintenance and evolution in open source projects distinctly.

As may be seen from this chapter, the industrial definition by IEEE of software maintenance and evolution quite differs from an academic definition of industrial software maintenance. This partly stems form the fact that industrial projects used to have a much clearer transition from development to maintenance. Often this transition included a change of departments and/or staff in a company, so the beginning of maintenance is clearly defined. However, due to modern software development methods and the nature of the project, e.g a web-project vs. an ERP, and the nature of project organization, e.g. waterfall development or agile, show the boundaries of a rigid approach as the IEEE definition of maintenance greatly. Godfrey and German as well as Bennett and Rajlich point out in their work that further differentiation is needed. In

the industrial case study in section 2.3, we pointed also out that modern software development processes converge and the maintenance process that is currently in place at the airport is largely identical to the development process during migration (albeit in a smaller scale).

Due to its ambiguity in terms, it is important to clearly define maintenance tasks as they will be later used throughout the thesis. We define maintenance tasks as follows:

Corrective maintenance are corrective measures to the code base that address errors[2], faults[3] or failures[4] (according to the IEEE glossary for software engineering [50]) in the code base. This includes preventive maintenance steps taken to address latent faults that are not operational faults yet. E.g. a regression bug that is found during testing after the software was released. This does not include preventive measures that improve non-functional attributes.

Adaptive maintenance are changes that affect the business logic of the system, e.g. changes to implement new or alter existing functional requirements. These changes may stem from changes to the model of the software, but also to alterations of algorithms.

Perfective maintenance are enhancements to non-functional attributes of the system, e.g. boosting performance, refactoring or improving system documentation

2.2 Commonalities of Software Development and Software Evolution

Before going into details on open source projects and the open source development process, it is important to define software evolution and software maintenance. The previous sections described a standard-based industrial take on the topic of software maintenance mostly from the point of view of a service provider. Software maintenance in an industrial sense used to be a clearly distinguishable phase in the software life-cycle, often marked by a transition from the product from the development team to the customer/user, but also by a transition from the development staff to the maintenance staff of a company. This definition of software maintenance was mostly an organizational definition of software maintenance. In a state of the art development process, the differences in the process itself between maintenance and development phase often blur. Historically speaking, the term software evolution was coined for the constant development of software past its incubation phase. However, from an agile development point of view this is

[2] The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition

[3] An incorrect step, process, or data definition

[4] An incorrect result

a fallacy, since agile practices emphasize continuous delivery, hence perpetual maintenance or development of features right from the start of development.

Godfrey and German remarked in [49] that industrial software maintenance and open source software evolution differ greatly. In their study on software maintenance of the Apache and Mozilla projects [51], Koponen and Hotti compare maintenance activities in both projects with the IEEE standard for maintenance (compare to section 2.1) and find that while there are parallels for some activities, there are also several differences, since there is no true retirement phase for the Apache and the Mozilla project currently. Neither is there a migration, in [51] this activity was replaced by release management. Unfortunately, their work does not dwell on the strategic differences between industrial maintenance and normal open source development, which were later suggested by Godfrey and German in [49]. Prior to their 2008 work, Godfrey and Tu in [52] published a case study on the evolution in open source software in 2000. This, together with Mockus, et al. in 2002 in [3], builds the groundwork for our understanding of software evolution in open source projects. However, as Godfrey and German already point out in [49], agile methodologies superseded our understanding of classical stages like evolution and maintenance in software development.

The Fundamental Open Source Software Development Process

In his groundbreaking essay "The Cathedral and the Bazaar" [16] Raymond described in great detail the paradigm shift in open source software development in the rise of Linux. Many of the notions described in this essay on the shift from the "cathedral" model of mature releases and small team sizes as well as a low number of communication channels to the "bazaar" model with large numbers of developers and innumerable communication channels have found their way into industrial software development as well.

Especially the proposition that "Given enough eyeballs, all bugs are shallow" can be seen in the industrial example in section 2.3. Several techniques such as constant delivery (early and swift user feedback), code reviews, pair programming, review meetings are typical success factors in modern software development. However, Raymond does not describe software development in an industrial setting. He describes software evolution, i.e. the constant change of a system from an existing shell to a full-fledged open source development project with active and numerous contributers as well as users. For Raymond, the open source software development life-cycle starts by extracting a usable scaffolding from an existing software that provides a groundwork of guidance of design and vision to nurture the evolution of the system. Figure 2.4 shows a simplistic view on lifecycle in open source and closed source projects. In this figure, the actual open source development phase starts after initial contribution of the existing scaffolding

system. It may be seen synonymously with software evolution.

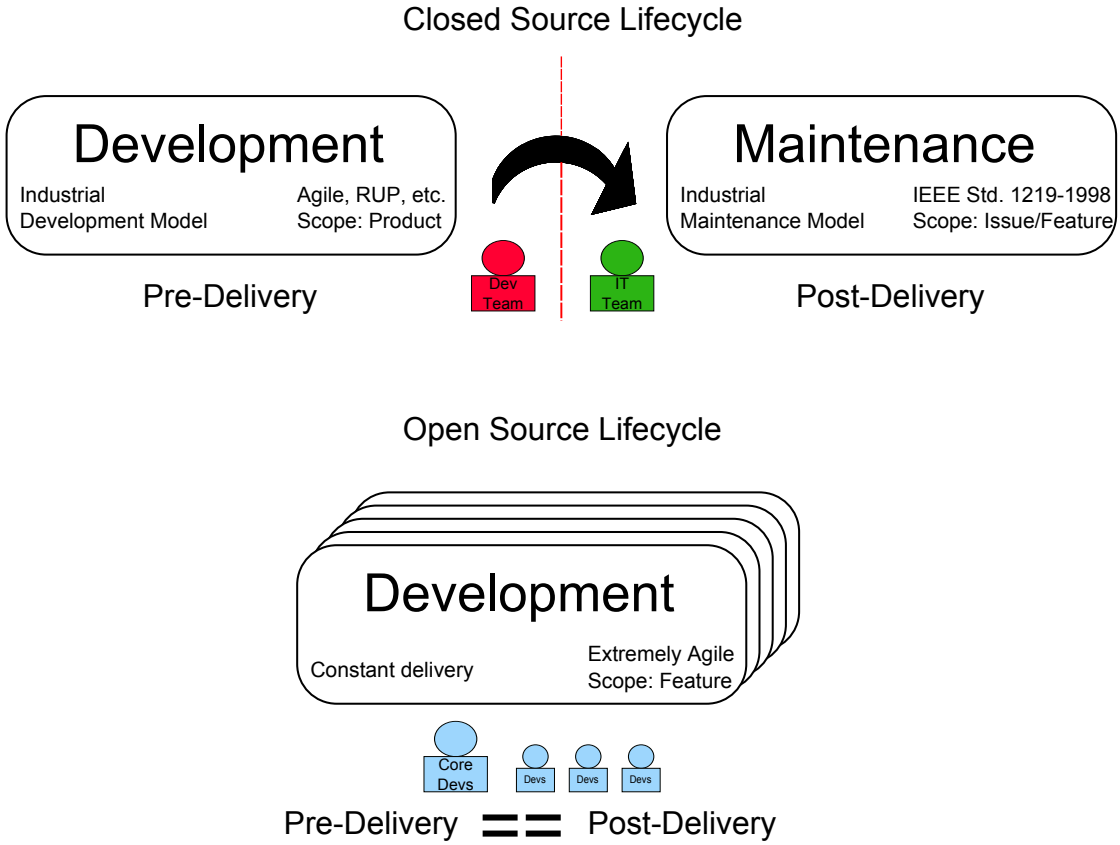


Figure 2.4:
Simplistic Overview of Open vs. Closed Software Lifecycle

While Raymond does not provide a full-fledged open source development life-cycle, two stages may be learned from his work. The initial phase, in which a runnable nucleus is provided by means of extraction or initial development, by sometimes only one person that has an interest in the system. At the end of this stage, the system is put out into the open, for people to participate as users or contributers. This has parallels to the classical approach on software maintenance, even though the focus is vastly different. This initial phase, the classical development phase, is, depending on the approach, very short, or at least vastly shorter than the maintenance phase. There might not even be development work per se involved. In the industrial setting, a maintenance phase would occur now and the IEEE maintenance model from figure 2.1 would apply. The system would be stable and finished and maintenance would cover only a small set of additional features (compare Reifer [53], where only one third of staff was dedicated to content for new releases in their survey). This is vastly different to the process that Raymond described

in his essay. The vast majority of features for the system would evolve post delivery. In fact, the term post delivery does not really apply to open source projects, as best practices are to release constantly, preferably daily. Post and pre delivery blur, the software is permanently available. This suggests a big discrepancy between the models for software maintenance described by Burton and later IEEE and software evolution, as described by Raymond. A lot of later work in the area of software evolution is based on Raymond's essay.

In the year after Raymond's essay, Godfrey and Tu in [52] observe two kinds of open source software. Some open source projects like Jikes or the Netscape web browser (the Mozilla project) were proprietary in-house development projects that were later on released as open source or under a similar license. Other open source projects like Linux, are created by single persons that extract some features from existing systems and start a new software from those existing parts. Godfrey and Tu distinguish between Open Source Software (OSS), which would be of the first kind described earlier, and Open Source Development (OSD) and an OSD model. This differentiation makes sense, since OSS might be developed in-house/on site, while OSD is truly transparent and collaborative from the very beginning.

They then point out the major differences between open source development and traditional in-house software development:

No commercial aim: developers in OSD projects are generally not motivated by commercial interest in the project, but particular personal interests

No schedule to adhere to: most developers in OSD projects have dayjobs or studies, so development cycles may be long and schedules are hard to enforce

Code quality: quality in OSD projects is hard to enforce, since code is contributed. On the other hand is code in open source visible and developers often take pride in their code

Immature code: developers may be eager to submit new features, before a certain code maturity has been reached. This may be addressed by proper configuration management

Planned evolution, testing and preventive maintenance: Code quality happens as a by-product of code reuse in other projects, there is no structured approach to preventive maintenance

Some of these findings are explained more cogently in Raymond's essay. However, they do point out differences between open source and industrial development. If we analyze these findings based on more current studies on software development and with the current penetration of agile methodologies in open and closed software, we find that the listed arguments are either not relevant because they apply to agile methodologies or are just fallacies that have been proven wrong in follow up surveys.

No commercial aim: while this is only partly true, see e.g. dual licensing concepts, or IBM's support of Apache, commercial aim has no impact on the development methodology itself, the process and the tasks undertaken

No schedule to adhere to: this is a fallacy and has long been proven otherwise - see e.g. [54]

Code quality: this is also a fallacy and has been proven otherwise - again, refer to [54] for an overview of open source software quality

Immature code: this is actually one of the key characteristics of both agile and open source development and arguably one of its largest advantages

Planned evolution, testing and preventive maintenance: as indicated in the previous section on industrial software maintenance, it may well be argued that these problems exist also in industrial software to a large amount

In another exhaustive study in 2002 [3], Mockus and Herbsleb examine the Apache and Mozilla development process. They define the following process steps for Apache and Mozilla:

Identifying Work to Be Done: Changes to the system may be proposed by anyone in both the Apache and Mozilla project. However, changes to the system are a privilege only for the core teams in both projects. In Mozilla, power may be delegated to module owners mostly, however the core team determines module ownership and resolves conflicts

Assigning and Performing Development Work: Mozilla uses Bugzilla to assign workload, when development work is done, developers are encouraged to mark an issue in progress, so no duplicate effort is wasted. Apache is a little bit more lenient, as developers may choose freely to work on their own enhancements or fixes and use various channels to distribute work load.

Prerelease Testing: Apache developers perform unit and feature testing to some degree on local copies, while Mozilla uses a more rigid test approach with dedicated test team.

Inspections: Due to their consenus based model, every change is reviewed by at least three Apache Group (AG) members. Additionally changes are distributed to the entire development community. In Mozilla, reviews are organized in two stages, one being the module level and the second one by a member of a so-called "super reviewer" group

Managing Releases: Responsibility for release management is rotated in the AG, while Mozilla has a dedicated group responsible for Milestone decisions

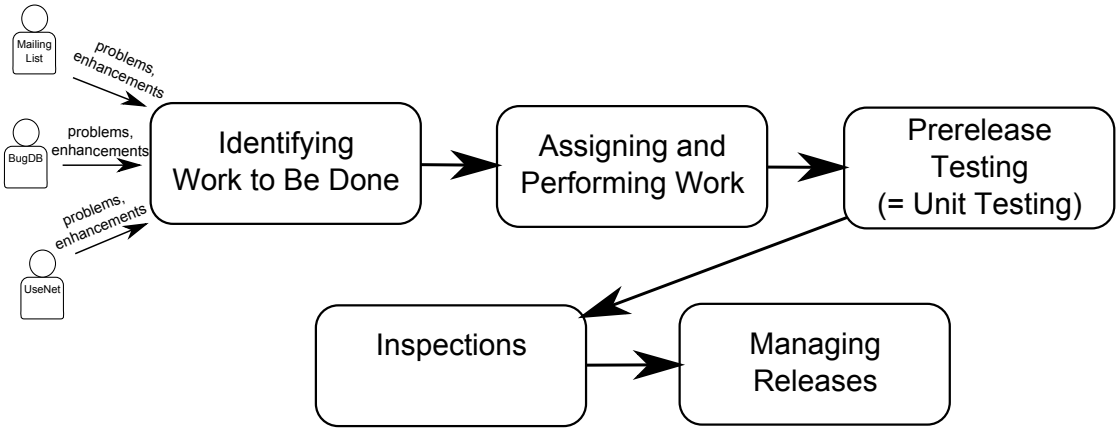


Figure 2.5:
Open Source Development Process of Apache and Mozilla in [3]

This structuring is in stark contrast to the findings of Godfrey and Tu in [52] as it actually bears similarity to current agile methodologies. This indicates that industrial software engineering and open source development might not be so different after all from todays point of view. In [55], Rigby et al. show how efficiently review processes are set up in open source projects and how rigidly applied. Interestingly enough Raymond's take on Linux development and current best practices in industrial software engineering practices have large overlap, as the findings of Mockus et al. also suggest. This strongly indicates that software evolution and state-of-the-art software development have much of their methodology in common. This makes sense, since there is a natural swap between developers in open source communities and industrial projects.

Agile Development in both Open Source and Industrial Context

In [44] Stefan Koch analyzes the parallels of agile principles and open source software development and proposes that similar empirical indications may be found in both methodologies, e.g. an "emphasis on highly skilled individuals .. at the center of a .. team, the acceptance and embrace of change by using short feedback loops with frequent releases .. and the close integration and collaboration with the customers and users". This indicates that open source and agile closed source development share more similarities than agile and waterfall closed source development. Thus, findings in the open source domain may carry over into the industrial domain.

The penetration rate of agile methodologies in the IT industry has been surveyed in recent years. In 2007, Begel et al. [56] published a survey of agile development at Microsoft and found that only 33% practitioners had picked up agile methodologies. This may be attributed to the fact that Microsoft as a large company has a more strict governance to satisfy, which is

problematic in the agile context (on the governance of agile projects in large-scale settings see e.g. Talby et al. in [57]). A more recent survey by Rodriguez, et al. in [58] reports a higher rate of adaption in the Finnish IT industry of agile methodologies. They found that 57.8% of their survey participants had picked up agile or lean methods for software development. Agile lifecycle management software vendor VersionOne even states in their most recent survey on the *State of agile development*[5] that 95% of the participants in the survey work in organizations that practice agile. While this may be a bit overenthusiastic, it is safe to assume that the number of agile adaptors has increased since Rodriguez study of 2012. Hence, the majority of software development efforts started recently are more likely to pick up agile development practices than not.

2.3 Case Study: Building Maintainable Software

This section is based on prior published work by Bernhart, Mauczka et al. in [4]. It shows how the industrial maintenance process begins to converge with open source software evolution by using shared best practices. It also describes some of the similar challenges in industrial and open source settings

A modern industrial software development project addresses the maintenance problem by providing a structured development method along with a tool chain and architecture decisions that support maintainable software. In [4] Bernhart, Mauczka, et al. describe the incremental reengineering and migration of a 40 year old airport operations system. The old, undocumented COBOL legacy system that could no longer be maintained had to be replaced by a java-based web application. The main reasons for replacing the old system were:

1. A new terminal was currently being build with new software requirements
2. The remaining two developers of the old system would retire in the next 24 months
3. License costs for the old platform

These main reasons for the migration of the system may be traced back to abstract and recurring problem fields of software maintenance:

1. Changing or enhancing the existing software is difficult, because:

[5] `https://versionone.com/pdf/VersionOne-10th-Annual-State-of-Agile-Report.pdf`

a) Changes require in-depth knowledge of the code base because the system has little to no up to date documentation

b) Changes to an existing critical software require large amounts of manual testing since there usually is no test automation at hand

2. Staffing for maintenance of the existing software is difficult, because:

   a) The target platform has in most cases reached its end of life and developers with the required skill set are hard to find

   b) Since little to no documentation is available, there is a steep learning curve to the old system that puts a strain on the existing resources

3. Costs for all maintenance related tasks are high, because:

   a) Legacy platforms may have very high license cost models, since open source platforms were uncommon then and while migration from a proprietary legacy platform to an open source legacy platform is not uncommon, it is also very expensive in itself and sometimes not even possible

   b) Staffing costs are high, because legacy applications are mostly based on platforms that are no longer being taught. This means a developer for the system will be hard to find and will be of high market value

High maintainability for a system as time-critical as an airport operational database (AODB) is very important, since outages of the software will result in large costs. To address this issue two approaches were used. The first approach focuses on providing procedures and methods to enforce the generation of quality code along exhaustive documentation. The second approach focuses on taking the right architectural decisions on the system design as well as on the component design. Before these approaches are discussed in detail, it is very important to give a short overview of the project team, as this is crucial for the later maintenance process of the AODB.

The project to migrate the AODB was put out to tender, since a large development team would be necessary for the migration, but later on, only a fracture of the team would be required for maintenance. So the core development team consisted largely of external developers, while the business analysis team was split evenly between internal and external business analysts. The quality assurance team was largely recruited externally, with some specialized internal testers for integration testing tasks. This setup is ideal to ease a transition from development phase to maintenance phase and is similar to open source projects after their incubation phase [16]. This is the foundation for a successful maintenance phase or stable software evolution and has been established right from the beginning of the project.

Architecture to Provide for Maintainable Software

The new architecture of the AODB uses largely open-source components and follows a thin-client approach. It provides a GUI frontend that is written entirely in HTML and JavaScript and communicates via REST calls with the server. These REST calls are consumed by web-services written in Java. These web-services correspond to the functional groups of the legacy system, e.g. there is one service to process departure of a flight. Communication between applications is handled via JMS - incoming messages trigger the same services that the GUI does. Outgoing messages by the AODB are published to JMS queues/topics to be consumed by other airport applications. The legacy architecture had also several hard-coded pieces of functionality that would change often, e.g. a notification system for departments to inform them of certain flight constellations and a user-based security mechanisms. See figure 2.6 for a diagram of the final architecture of the AODB.

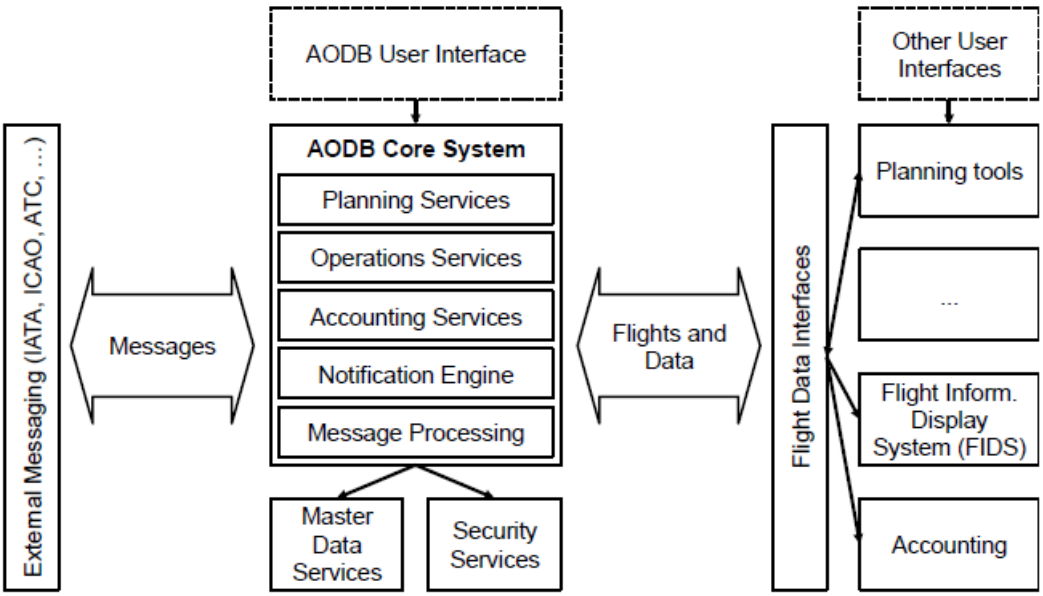


Figure 2.6:
Maintable Architecture of AODB in [4]

This architecture proved to be highly maintainable for a number of reasons:

1. Loose Coupling between front end and back end provides separate testability and deployability, which allows fast bug-fixing cycles in production

2. Clearly defined interfaces allow for a high degree of test-automation in the back end. This

allows a high change flexibility for the maintenance phase, since bug-introducing changes can be detected swiftly and early

3. Staffing for the used technologies is easy, since Java is a very common programming language

4. Open source components keep the licensing costs to a minimum

5. The hard coded notification system was replaced by an event-driven rule-based notification system. Its configuration may be changed at run-time by using a Java Expression Language (EL) in the database

6. The hard-coded security and tenancy checks were replaced by a declarative Java annotation-driven approach that increased readability and modifiability

Aside from being highly maintainable, this architecture proved well-suited for integration with the 20+ other airport applications as well.

Methods and Processes to Provide for Maintainable Software

A maintainable system consists of more than well-readable code. Besides an architecture that enables maintainability, it also requires procedures that allow developers to understand the dependencies of the code and to ease the estimation of impacts of a change to the system. This is similar to software evolution in open source projects. It is important that knowledge of the system is shared in the whole team and that functionality is not bound to certain project members. There are a number of measures that have been undertaken by the project team in [4] to achieve certain maintenance goals.

1. Using SCRUM as a development method achieves the following goals:

    a) Collective code ownership - this means that the code is by itself understandable, but also that the code is understood by the whole development team including future maintenance staff

    b) Including software quality gates in the Definition of Done [6]

2. Providing a high degree of test automation achieves the following goals:

    a) Fast feedback for developers on potential bug introduction by providing a very high coverage of unit tests

[6] https://www.scrum.org/Resources/Scrum-Glossary/Definition-of-Done

b) Provide confidence to be able to perform large changes to backend functionality by writing a high number of integration tests

c) Allow GUI-component changes by having a large set of automated end-to-end tests

3. Writing detailed documentation and keeping it up to date achieves the following goals:

   a) Enables fast ramp up times for new staff

   b) Fast response times to complex bugs, since functionality is well documented and can be understood easier

   c) Project members that were not directly involved during the development phase are still able to perform maintenance

Conclusion

Maintainability as described in the case study may be used synonymously with evolvable software. The problems described are common in legacy applications and often signal the end of life for an application. Software evolution was no longer possible in the old software. To avoid these pitfalls and to mitigate the transition from the initial development to software evolution, the processes were not changed between development and evolution, only scaled down to a smaller team size. Quality assurance processes and delivery processes remained the same. This is a strong argument for the parallels between agile software development and software evolution and thus the assumption that agile software development and open source software development share similar processes.

Some of the previously listed aspects are already prevalent in open source projects. Collective code ownership is largely common (see Mockus et al. in [59], as are fast release cycles (see e.g. [52]). Differences between industrial and open source projects concern test practices, which are more ad-hoc organized (see also [52]) in open source projects and happen at unit level rather than on a system testing level[7]). Instead, open source projects use complementary quality assurance methods, like feedback by their users on unstable versions, e.g. Linux uses branch management (development and stable releases) to provide versions to-be-tested for early adopters.

[7] System testing as defined by the ISTQB foundation level syllabus (chp. 2.2.3 p. 26) - http://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html4

2.4 Organizational and Social Aspects of Software Evolution

Lientz et al. in [48], aside their findings on user understanding, identified motivational factors of the software developers as a crucial impact on maintenance performance. They point out that programmer effectiveness (which is determined as productivity, motivation and qualification) has a large impact on efficacy during software evolution. Chen et al. in [60] identify personnel resources problems as one of five aspects impacting software evolution and rank project turnover as eighth of their top ten issues during software evolution (the other issues being related to code quality, documentation and requirements). Consequently it is important to monitor long term motivation for developers. Similarities between an industrial setting and open source settings have been observed (e.g. by Paulson in [61]), even though no studies could be found that compare motivational factors between industrial and open source settings. However, as has been indicated earlier, findings may carry over from closed to open source projects.

Developer Motivation in Industrial Projects

Hall et al. in [62] present a survey of 92 studies on motivational factors in software development. They differentiate between intrinsic and extrinsic factors, as well as between inherently software engineering related motivators and generally work related factors. Life-cycle models are e.g. an inherent software engineering and intrinsic motivator, while good management and sense of belonging are extrinsic factors. While some motivational factors are unique to the industrial settings (especially those factors that cover basic needs, e.g. job security, work/life balance), the majority are shared between industrial and open source projects. Hertel et al. in [63] support this notion by stating that intrinsic and social comparison motives were the main motivations in their survey on the contributors to the Linux kernel. Table 2.1 lists extrinsic factors for developer motivation based on the survey presented by Hall et al. in [62] and shows that most of the identified extrinsic factors could also apply to an open source setup.

Organizational Structure and Developer Motivation in Open Source Projects

In his study on shared leadership in the Apache project [64] Fielding describes the organization of roles and responsibilities in the Apache Group (AG). Understanding how open source development teams are structured is important to understand the process, but also to infer contributor motivation as well as communication channels. Fielding describes "[..] a system of voting via email that was based on minimal quorum consensus". Every developer was allowed to cast his vote on the mailing lists, but only AG member votes were binding. A minimum of three positive votes and no negative votes were required to submit a change to the final code base. This

Extrinsic Motivators
Good Management
Sense of Belonging
Rewards and Incentives
Feedback
Job Security
Good work/life balance
Appropriate working conditions
Successful company
Sufficient resources

Table 2.1: Extrinsic Motivators in Industrial Projects

meant that not all AG developers had to be involved in all issues, but it also lead to at least three code reviews on any changes to the system. To be a member of the AG, a frequent contributor would be nominated by one member and had to be approved by the voting AG members. In their study of the Apache project in [3], Mockus et al. states that the AG started with 8 members and reached 25 in 2002.

Mockus et al. also describe the organizational structure of the Mozilla project. In 2002 there were 12 members of the mozilla.org staff. Their main duties were to "[..] coordinate and guide the project, provide process and engage in some coding". This particularly means community and interaction tool grooming, structuring of development work, etc. Decisions are delegated to "[..] individuals in the development community who are close to that particular code.". Developers with a track record of quality code may be granted commit rights on the CVS repository. Ultimate decision-making is up to the mozilla.org core team though and how consensus is found inside the group is not found in their study. The organization is vaguely similar to the Apache organization, as there is a core team of developers that decides on issues, but Mozilla's organization delegates decisions to a module level, where the module owner calls the shots, while all changes in the Apache project are voted on by possibly the whole AG.

Further research on the organizational structure in open source projects by Bird et al. [65] reveal a more in-depth picture of the organization of open source projects by shedding some light on how informal organization is happening in large open source projects. They establish that sub communities in the analyzed projects exist and that modularization of these communities is stronger when the discussion is product-related. Developers in these sub communities will have more interaction in their development work than with developers outside their sub communities (see figure 2.7).

Hong et al. in [66] compare developer social networks (DSN) and general social network (GSN). In their study they show that while DSN and GSN have some similarities, DSN have

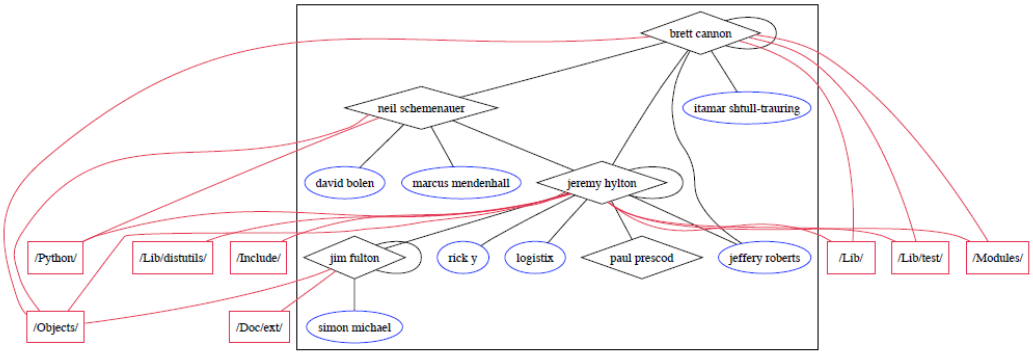


Figure 2.7:
Development Sub-Community in Python from April to June 2003. Diamonds are Developers, Ovals are Participants and Rectangles are Directories committed to as Presented by Bird et al. in [65]

some unique attributes. DSN do not seem to follow a power law degree distribution as GSNs do. They also find that both DSN and GSN are prone to build community structures (which supports the findings in [65]), however no comment is given on the fact that DSN should have a stronger community building drive due to shared work on software modules. They also find that influential people often affect other developers in the DSN. They go as far as suggesting the departure of an influential developer may cause an exodus in the development force.

Lin et al. in [5] describe the following community evolution patterns:

One-to-one derivation: One community evolves into another, this includes shrinking and expanding

Merge: Two distinct communities merge into a new community

Split: One community splits into two distinct communities

Extinct: A community ceases to exist

Emerge: A new community starts to exist

Hong et al. use these patterns in the DSN context as argument for their theory how influential community members may impact the development force. In related works, Nia et al. point out the potential pitfalls in network analyses in [67], while Sarma et all. provide Tesseract, a visualization tool that provides insights into social (e.g. community interactions) as well as technical (e.g. developer to artifact) relationships.

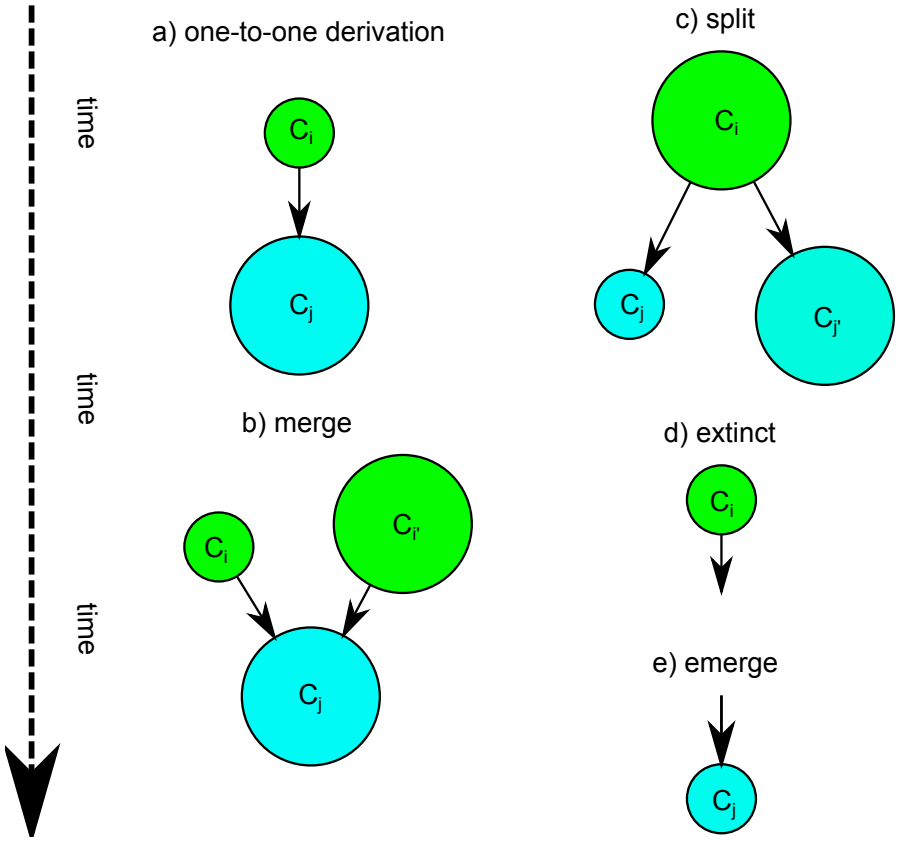


Figure 2.8:
Community Evolution Patterns by Lin in [5]

Tsay et al. in [68] show that not only technical facts but also the social standing of a potential contributor play a key-role when contributions are accepted into a code base. This is a noteworthy aspect when discussing developer communities as introduced in [66] and shows the need to further analyze the social aspect in open source development.

Crowston et al., in their literature survey on open source development [69], state that while economical reasoning behind joining an open source project has largely not been discredited yet, supporting research is hard to find. They report that studies generally show three types of motivation, which are extrinsic motivation, intrinsic motivation and internalized extrinsic motivation. The most frequent extrinsic factors are reputation and career development. Intrinsic factors are fun and sharing or learning opportunities, while user needs are the most commonly mentioned internalized extrinsic motivations.

In his paper on motivation [70], Shah tries to find a full set of developer motives in open source projects. He identifies two archetypes of open source participants, the need driven par-

ticipant and the hobbyist participant. The need driven participant joined the project because of the need for the software to perform a task (which is commonly work-related). These type of participants opt for open source so "[..] they could view and change the code to best fit their own needs". When they performed their need-induced change, they contribute their code for several reasons. These are reciprocity, future product improvements, desire to integrate own code and career concerns. However, these factors do not seem to generate more long-term participation. Hobbyists on the other hand, stayed for fun and enjoyment which are reportedly greater motives for long-term participation. This is a major factor to consider, since freedom of schedule and creativity therefore seem to play important roles in continuous contribution to an open source project. In contrast to this observation, Shah points out that classical maintenance chores are also done by the hobbyist contributers. This is interesting as one would assume that chores would relate to extrinsic motivators only available in industrial settings.

Wu et al. in [71] confirm some of the hypothesis by Shah and point out that developers are motivated by helping, but also economic incentives, related to enhancing human capital, career advancement and personal requirement in software development. Interestingly enough, fun or the hobbyist as indicated by Shah does not seem a factor in Wu's study. Fang and Neufeld in [72] point out that it is important to differentiate motivation to participate in OSS projects over time, i.e. motivation for participation changes from the short to the long term. They apply the theories of legitimate peripheral participation to show that social interaction and motivation over time are important factors in OSS participation. They argue that situated learning and identity construction as elements of social interaction are important for sustained participation. This is similar to the findings of Sha in [70] in so far that short-term participants were need driven and long-term dedication was enjoyment driven. Fang and Neufeld expand on this notion by proposing that long-term participation is a result of repeated positive situated learning and identity construction social interactions. Fang and Neufeld also discovered that only participants with a mixture of development and conceptual work as their activity profile were successfully integrated into the community. Initial motivation and access to the project, while necessary, are not key to successful continued participation according to them.

2.5 Mining Software Repositories and Natural Language

Repositories offer an abundance of data on what happens during the development of systems. There are version control systems (VCS), i.e. source code repositories (CVS, GIT, etc.), issue trackers, i.e. bug and feature repositories, mailing lists, code repositories (e.g. Sourceforge[8]) and more. All these repositories provide data on the software engineering process in the large as

[8] `http://sourceforge.net/`

well as in great detail. While there is a lot of data available because these repositories are integrated into the daily development life, there are pitfalls in mining these. Bird et al. in [73] show that considerable care is necessary when analyzing SVN or GIT, since there are differences how centralized and distributed code repositories handle changes. In related works, Kalliamvakou in [74] show that not only VCS platforms have their pitfalls, but also that hosting platforms like GitHub[9] have perils as well, e.g. not every repository hosted is a project, most projects are inactive, some projects are not even software development projects. One community especially dedicated to the research in software repositories is the "Mining Software Repositories" community. The MSR conference[10] in 2013 brought the 10th anniversary of the conference.

The first international Workshop on Mining Software Repositories was held in 2003 as part of the International Conference on Software Engineering (ICSE). In 2008 MSR became a Conference in itself, but was still co-located with ICSE. The MSR Conference hosts challenges each year where researchers may submit analyses of defined data sets, a short paper track and a research paper track. Overall, MSR research is of course not only happening at the MSR conference, however the conference proceedings give always an interesting insight on the current topics of interests and future research areas.

Hemmati et al. used the anniversary as occasion to review all full papers that have been published in the last decade at the MSR conference. They identified 28 categories of research, which they merged into high-level themes. These themes coincided with the typical MSR process, which may be seen in figure 2.9.

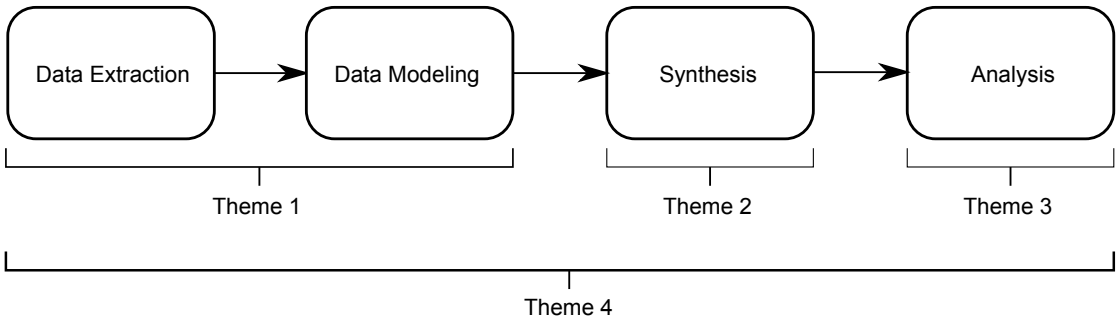


Figure 2.9:
Steps of a Typical MSR Process in [6]

In another paper dedicated to the anniversary, Demeyer et al. [75] present interesting findings on the historical research focuses of the MSR community. They find that the topics change and software evolution are in the focus of papers in the last decade. They notice an influx in "[..]

[9] https://github.com/
[10] http://2013.msrconf.org/

research studies with reference to 'metric' and 'test'" while "design pattern" seems to be on the decline. The most frequently analyzed project is the Eclipse project, the Linux kernel and Mozilla are also relevant. Since it is much easier to access open source projects, these are the dominant research targets. In the beginning of the MSR, CVS and Bugzilla were the major repositories to be analyzed. Bugzilla is being replaced by JIRA however, CVS has been replaced by Subversion, which in turn is being replaced by GIT. In general, source code repositories are the most prominent target platforms. However, they also identify a trend to other sources of rich data, e.g. archived project communications like Stackoverflow[11]. In their opinion future research should focus on commercial and closed-source systems. They also stress that the community is prone to react slowly to emerging platforms (e.g. JIRA and GIT). And finally they propose that research in MSR deliver more tangible results for practitioners.

In 2008, Hassan wrote a survey of challenges for toolkits in the MSR field [76] and identified the following major challenges for MSR research:

Simplifying the Extraction of High-Quality Data: The heuristics used by researchers need to be examined and documented carefully and should be offered as tools with high usability for untrained users.

Dealing with Skew in Repository Data: Repository information is often subject to noise and skew in it. This means that "[..] more robust algorithms and data re-sampling techniques should be adopted"

Scaling MSR Techniques to Very Large Repositories: Data sources grow in size over time, hence high performance tools and techniques are required to keep up with the growing information pool

Improving the Quality of Repository Data: Hassan suggests that guidelines for practitioners willing to help in research are handed out to ease analysis of projects

These findings provide a good starting point for future toolkits in the field of software evolution research, which the MSR community is a specialized part of. Hassan encourages the use of more than one repository type when doing research since this grants a more complete view of the project. He also stresses the importance of "showing the practical benefit of MSR techniques". Hassan also suggests to broaden the research area by moving beyond code and bug related repositories by exploring non-structured data like communication archives, but also non-structured data in classical repositories. An especially interesting topic of research in MSR therefore is in natural language processing to use unstructured data to derive information.

[11] `http://stackoverflow.com/`

Natural Language Processing (NLP)

Natural language processing (NLP) has been a prominent research topic in the last years in the MSR community. Hassan in his survey [76] points out the importance of leveraging non-structured data. In their 2008 survey of software evolution [49], Godfrey and German point out that software systems tend to be very different even when considering the same type of software system (e.g. ERP systems). This technical heterogeneity is also an indicator that an approach that leverages meta information on software artifacts is a logical next step in the analysis of software evolution and maintenance. They identify the need to link software artifacts together as a future challenge for research in software evolution (an example of how this is achieved by using natural language processing is given in chapter 8). Hindle et al. in [29] go so far as to proposing, while natural language may be diverse and potentially complicated, "[..] what people write and say is largely regular and predictable". They then apply this proposition to source code, to show that source code, while potentially complex and mind-boggling, is in the large repetitive and simple. This makes sense for OSS especially, where code is written that needs to be understandable by the public. They leverage this approach to build an Eclipse Plug-In for code completion and suggestion. Allamanis and Sutton in [77] extend this approach by expanding the training projects vastly. Allamanis and Sutton in [78] process Stack Overflow questions to categorize questions according to concepts and in the next step, they categorize questions by type, e.g. instead of questions on the concepts of "games", they categorize questions by the kind of information that is requested, e.g. build issues. This way they identify question types that do not vary across programming languages.

An interesting study was performed by Guzzi et al. in [79]. In this study, Guzzi et al. analyze communication in OSS mailing lists. They find that implementation details are only discussed in 35% of the threads in the lists and moreover, core developers participate in less than 75% of the threads. This seems in accordance with the findings of Fang and Neufeld in [72] mentioned earlier that indicated how core developers both develop AND do conceptual work.

Kevic and Fritz present a novel approach in [80] where they implement a Natural Language to Source Code Language dictionary. Their idea is to map natural language terms to source code elements, so when a change task is described, the dictionary is capable of suggesting the proper class or method that the change is related to. Merten et al. in [81] try to automatically distinguish between technical information and natural language text, which is an important task, since any NLP effort on repository items containing large bodies of text need to be able to either ignore or deal with large parts of technical information. This is especially interesting in conjecture with Kevic's work on the NL and SCL mapping dictionary. Merten et al. apply heuristics like detecting regular expression patterns, counting of similar lines, etc. as well as a clustering

approach to identify technical information pieces. Efforts in this area of data extraction also include work by Bacchelli et al. in [82] and in [83] on how to retrieve technical information from emails and on how to extract technical information from mailing lists. Bettenburg et al. in [84] also write on general separation of natural language and technical text.

Another practical application of NLP techniques is presented by Wang et al. in [85]. Wang et al. use natural language and execution information to detect possible duplicate bug reports. They present a two-step approach where they first calculate Natural-Language-based similarities and then execution information based similarities between a new bug report and existing ones that shows some success in duplicate report identification. This also suggests the beneficial usage of NLP tools on natural text bodies as well as on technical text bodies. Wang et al. evaluated their approach on Eclipse and later on Firefox and showed promising results. A major drawback of their approach is the reliance on execution information - this might not be available especially in long-running projects.

Amor et al. in [86] use methods of NLP to normalize commit messages for the FreeBSD project. They then use Bayesian classifiers to categorize commits. However, their results are neither fully automated nor do they achieve a high success rate, mainly because of the bayesian classifiers and a seemingly unorthodox categorization per se of maintenance tasks that is loosely based on Swanson's categories and has been extended into various sub-categories. Based on the work by Amor et al., Hindle et al. in [87] use different machine learners to automatically classify large commits based on a prior manually classified training set. The machine learners are able to consistently classify commits into maintenance categories, however, Hindle et al. point out that the author identity also plays an important role for large commits. It seems there are "specialized" authors for certain maintenance tasks.

In their MSR Cookbook paper [6], Hemmati et al. summarize suggestions for mining software repositories. An important suggestion for NLP is 'plain text requires splitting, stemming, normalization and smoothing before analysis'. They further state that 'text analysis should be manually verified in addition to regular bias reporting'.

Classification of Software Artifacts

There are several approaches to build automatic classification systems for software artifacts. Some of these approaches use machine learning techniques to automatically classify software artifacts. Antionol et al. in [88] show how to apply the WEKA tool[12] to build three of these classifiers to detect bug or non-bug issues in a BTS by analyzing the issue's available bodies of text. They compare different classifiers, namely alternating decision trees (ADT), naive Bayes classi-

[12] http://www.cs.waikato.ac.nz/ml/weka/

fiers and logistic regression. Machine learning techniques are differentiated by the approach to training data they use. There are the following families of machine learning techniques:

- Unsupervised Learning: The algorithm needs to find his own structure in the data given
- Supervised Learning: A labeled sub-set of the data under consideration is given and the algorithm has to classify the other data with a low error probability
- Reinforcement Learning: An incrementally learning approach for classifiers that is based on learning from user behavior

The techniques by Antionol et al. in [88] are supervised learning techniques. They achieve precision between 64% and 98% and recall between 33% and 97% depending on project and machine learner used with a correct decision rate as high as 82%. They argue that out of 1.800 manually-classified issues, less than half are related to corrective maintenance. This points us to implement finer-grained classification for BTS.

Kim et al. in [89] use machine learning classifiers to determine whether a change is buggy or clean based on similarity to previous buggy or clean changes. They use information gathered from VCS. They train a Support Vector Machine classifier on 12 open source projects. Their corpus is based on self-classified bug fix changes based on their commit messages. Once they identify a bug-fixing change they trace backward to identify the bug-introduction change similar to [90]. They also extract sets of meta-data on all changes (e.g. change time, size, author, etc.). All this information comprises the corpus, which they then use to train a classification model. Once their classifier is trained, any new change may be fed to it for classification. They achieve results between 0.43 to 0.86 recall and 0.44 to 0.85 precision for identifying buggy changes. Kuhn presents a lexical approach to label software components in [91]. They use log-likelihood ratios of word frequencies to automatically provide labels for components.

De Lucia et al. in [92] put the worthwhileness of using IR methods for source code artifact labeling in general to question and arrive at the conclusion that complex and bloated approaches fail to match the simplistic logic in human labeling of said artifacts. Their results are interesting for a classification approach as well, since labeling is just a form of classification. Our findings also carry the notion that a simple approach that is focused on primitive, atomic items provides better results than overloaded approaches like the Latent Dirichlet Algorithm [93]. Herzig et al. present a survey on five open-source projects and more than 7.000 issue reports, where they manually classified these reports and found out that 33.8% of all reports on bugs were not bugs at all but missclassified. On average they find “[..] that 39% of files marked as defective actually never had a bug”.

Mockus and Votta in [18] on industrial projects as well as Hassan in [19] on open source projects use commit messages to classify changes to the code base into Swanson's maintenance categories and achieve potential-bearing results. One part of the SubCat methodology is based on customizable NLP-techniques, which uses an extension of their presented approaches. These efforts on classification using the SubCat methodology have been published in [38], [10] and [11] and are presented in part III. Fu et al. present an improvement to SubCat in [94], where they employ semi-supervised Latent Dirichlet Allocation based on the cross-project valid dictionary from [11] and present promising results. In 2008 Hattori and Lanza in [95] classify tasks in open source development based on their own devised task framework for open source development tasks. Their reasoning is founded on the difference in industrial software maintenance and open source software evolution. This is counter-intuitive to the findings in recent studies of the parallels between industrial and open source projects and thus discarded for the work at hand. Furthermore, their simplified classification algorithm has been found to deliver lacking results in a recent reenactment of the study by Fu et al. in [94].

In [96] Hindle et al. use a large set of available machine learners of the WEKA tool to extract relevant topics for software evolution analysis, using a cross-project taxanomy. They propose to label topics by non-functional requirements, since they assume that non-functional are cross-project valid. They use three different methods for topic labeling: semi-unsupervised labeling of topics, supervised labeling of topics with a single NFR and supervised labeling of topics with multiple NFRs. Similar recent work on the topic of machine learners may also be found in [97] (building topics for FAQs), [98] (topics in android bug reports), [99] (matching topics extracted from VCS against high-level requirements), [100] (using topics to avoid bug-report duplication) and Scott et al. use LDA in [101] to find topics in source code analysis.

Sentiment Analysis

Sentiment analysis is just turning up as a topic of interest in software evolution research. It seems like a logical next step, once developer motivation and social structures have been somewhat staked out, to look into developer emotions next. Earlier in this chapter, we already indicated by prior research how important motivation is for software maintenance in industrial and for software evolution in the OSS context. Motivation and emotion are closely related to each other, hence looking at tools for automated "emotion mining" to apply in the research of software evolution is obvious.

Murgia et al. in [30] try to manually mine software artifacts for emotions. They examined 792 developer comments of the Apache projects using an emotion framework consisting of 6 major emotion categories (love, joy, surprise, anger, sadness and fear) to find out whether

absence or presence of emotions in issue reports may be consensually be determined by human raters. They also researched how additional information on an issue affected the rating. They argue that emotion mining in software engineering could bring insights on emotional triggers in open source projects, like releases or recurring bugs. They found that agreement on all 6 emotion categories by both raters only occurred on average about 46% on the comments. The highest agreement could be found on the absence of emotions in an issue report. The highest agreed upon emotion was love. Adding additional information to the rating process actually lowered agreement rates. Murgia et al. suggest to use only `love, joy` and `sadness` for automated emotion mining, since these provided the higher rater agreement.

Before Murgia's fundamental work, Gomez wrote an interesting article on a simple emotion mining experiment on GitHub [7] by using regular expression filters for certain key words. He found out that Perl contains many commit messages that express surprise, while VimL (see figure 2.10 seems to aggravate its developers mostly. His approach is a simplistic one by using lists of keywords, similar to our lexical approach for maintenance activity categorization in chapter 3. His article is certainly a trigger for further research, which was picked up by Murgia.

Guzman et al. in [31] propose an approach to increase emotional awareness in development teams by providing "quantitative emotion summaries". They analyzed three development teams over three months and lead interviews to find correlation with their automated assessment. They use latent Dirichlet allocation (LDA) for extracting topical information and lexical sentiment analysis to assign an emotion score. Unfortunately, their work is still in an initial state and was only performed on a very small scale, still interviewed project managers agreed on some emotional findings as well as on the potential usefulness of their approach. Guzman et al. also provide a study in [102] where they performed Sentiment Analysis on commit messages and found that commits made on Monday seem to elicit negativity in tone. They further state that Java projects tend to have more negativity in their commit messages. However, the study is an initial study and they suggest the need for a more representative sample.

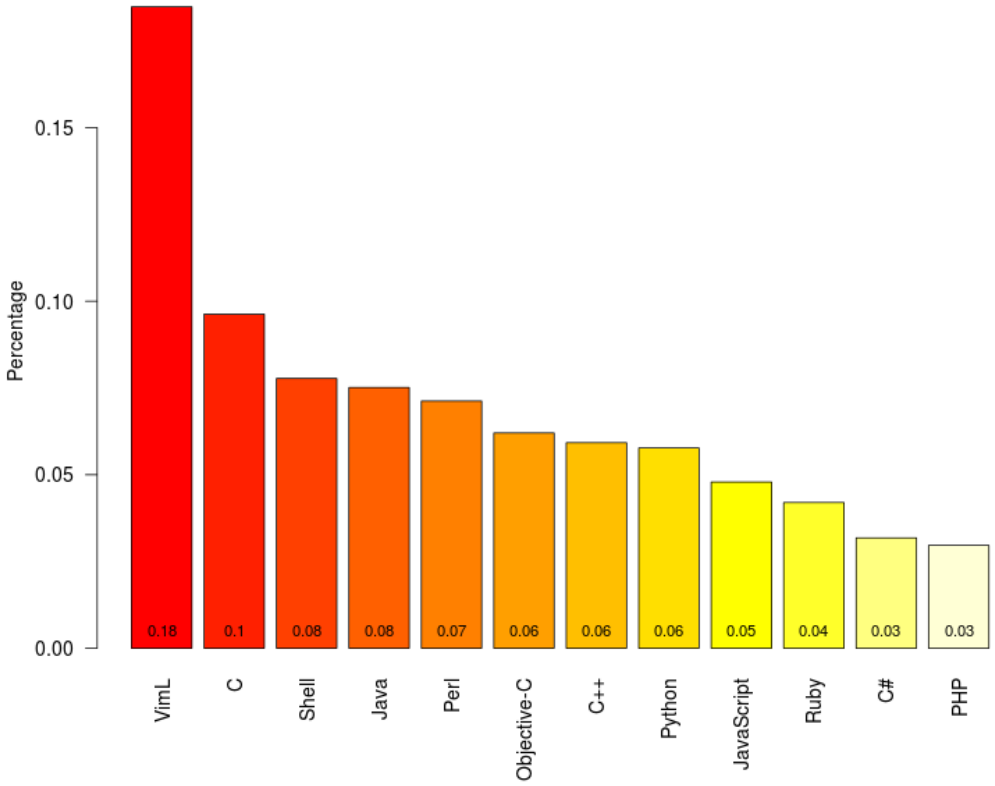


Figure 2.10:
Percentage of Commit Messages with Expressions of Anger - Emotion Mining on GitHub [7]

Part II

Design

CHAPTER 3

Designing a Framework to Use NLP Techniques for Data Mining in Software Repositories

Contents

> Simple analyses often outperform their complex counterparts.
>
> The MSR Cookbook [6]

The previous part contained the theoretical foundation for the innovative methodology, which will be described in the following sections. The focus of this methodology is to leverage natural language in software related artifacts to gain insights into software evolution. As pointed out previously, research in NLP is important in the field of mining software repositories and much knowledge can be gathered from meta-information in natural language compared to source code analysis alone. The intent behind an action in software evolution is easier and simpler expressed

in natural language than by re-engineering intent from e.g. source code. Hindle et al. in [29] go so far as to proposing, while natural language may be diverse and potentially complicated, "[..] what people write and say is largely regular and predictable". This notion encourages the design of mining tools to apply NLP techniques on repositories.

We therefore propose the SubCat methodology to leverage NLP techniques for the mining of software repositories. It centers around a state-of-the-art mining process and is based on the belief that the integration of NLP techniques in software evolution research is important and will deliver insights that may be usable in the broader field of software engineering.

The following chapter is therefore structured as follows:

- Section **Challenges in Software Evolution** provides a list of challenges the research field of software evolution poses and the SubCat methodology shall address
- Section **Design of Robust Mining Tools** shows the concepts behind a robust mining tool that may easily be extended and configured
- Section **Integration of Analysis Tools** explains how different modules are used to integrate various analysis tools at the defined stages of the mining process
- Section **Presentation Layer** describes how the presentation layer encapsulates presentation logic for the various output channels of the implementation
- Section **Design of a Framework to Use NLP-techniques in Software Evolution Research** combines all three aspects into one framework that uses a layered architecture to provide separation of concerns and an adequate design to implement the devised mining process

3.1 Challenges in Software Evolution Research

Based on the surveys of current software evolution research by Demeyer et al. in [75] and Hassan in [76], we identified the following research issues of current software evolution research that the SubCat methodology will address:

- **Challenge 1:** Provisioning of tools to analyze active research topics - [75]
- **Challenge 2:** Integration of popular and emerging mining infrastructure - [75]
- **Challenge 3:** Applicability of the methodology for management - [75]
- **Challenge 4:** Simplification of extraction of high-quality data - [76]

- **Challenge 5:** Scaling MSR techniques to very large repositories - [76]

In the following sections, we will describe the challenges and the contributions of the SubCat methodology to tackle them.

Challenge 1: Provisioning of Tools to Analyze Active Research Topics

Demeyer report in [75] that current research trends focus on "metrics" and "test", while "text mining" in 2013 rarely appeared. The SubCat methodology shall provide a framework to support usage of various metrics to provide management with a project cockpit-like functionality based on text mining as a novel approach (a potential field of research according to Hemmati et al. in [6] among others). As metrics have been identified as an important goal by the research community and the importance of data to improve development processes is evident for industrial scenarios, the ultimate goal of the SubCat methodology is to provide a platform to implement metrics that may be used by management as well as researchers. The SubCat methodology shall provide capabilities to measure various characteristics of a project, e.g. based on existing metrics like newly opened bugs, newly closed bugs and number of active users, or novel metrics like distribution of development tasks, social interaction patterns and sentiment expressed in VCS and BTS. The SubCat methodology is extensible, so a subset of these metrics, defined by possible application scenarios of SubCat (see part III), shall be implemented to proof that the SubCat methodology is able to provide appropriate tools.

Challenge 2: Integration of Popular and Emerging Mining Infrastructure

Demeyer et al. in [75] found CVS and Bugzilla to be the most frequently cited VCS and BTS. While the most cited, CVS is no longer the most popular VCS according to the popular open source software development indexing site `http://openhub.net`, which indexed over 600.000 projects in January 2016. The most popular VCS[1] are Subversion (47%), GIT (39%), CVS (9%) followed by Mercurial (2%) and Bazaar (1%). Unfortunately, no similar statistic may be found on BTS, though Demeyer suggest JIRA as an upcoming new BTS platform, which is also the impression of the author based on experience in industrial projects. The SubCat methodology should be able to integrate emerging mining infrastructure. However, not only mining infrastructure is subject to popularity, so is the programming language. The SubCat methodology shall be robust to both of these aspects. It shall take integration of mining infrastructure into consideration for its architecture as well as be independent of the used programming language.

[1] `https://www.openhub.net/repositories/compare` - last visited on 24.06. 2016

Challenge 3: Applicability of the Methodology for Management

Aside from data that is important to project management, it is also paramount that the applicability of the methodology is given (also identified by Demeyer et al. in [75]). This means it shall be easy to use, customizable and transferable between projects. Adhering to best practices is one part to achieve user acceptance, another is to provide functionality in the proper channels. The SubCat methodology shall provide different types of output channels to cater to diverse stakeholder needs.

Challenge 4: Simplification of Extraction of High-Quality Data

A more specific aspect of applicability is the simplified extraction of data as pointed out by Hassan in [76]. Using NLP-techniques to extract high quality data is largely simplified by using existing NLP frameworks or the implementation of evaluated NLP-techniques like the identification of technical information in bodies of text. The SubCat methodology shall provide a framework that allows a simplified extraction process, where raw-data, which may be used for further analysis by techniques like machine learners or self-designed classification schemes, is stored and kept separately. It shall also provide an analysis process that is easy to use and provides data also in a channel fitting for the corresponding user group.

Challenge 5: Scaling MSR Techniques to Very Large Repositories

A more technical issue that is one of the reasons for low user acceptance in MSR techniques outside the research community is that many MSR techniques do not scale to very large repositories according to Hassan in [76]. The SubCat methodology needs to take this into consideration and shall implement a scale-able framework.

3.2 Design of Robust Mining Tools

Before we delve deeper into the NLP functionality offered as part of the SubCat methodology, it is required to think about the steps that are necessary to provide the raw data the NLP functionality can be applied on. This fundamental mining process for SubCat is derived from best practices and implements the mining process as summarized by Hammati in [6]. Consequently, the framework implements four required steps of the *fundamental mining process*:

1. **Configuration:** This step consists of configuring project and user meta-data, e.g. BTS endpoints, VCS URLs or credentials

2. **Mining/Pre-processing:** Miners connect and aggregate data from targeted BTS and VCS and offer potential extension points for other repository types

3. **Classification and Sentiment Analysis/Post-processing:** During post-processing, interesting features of the aggregated data are mined and data is linked. This step may include, aside of NLP techniques, classification of artifacts and the linking of accounts between repositories and other functionality

4. **Data Exploring/Viewing:** Finally, a user can use graphical tools for data exploration and viewing

Step 1 and 2 are usually done once per project, while 3 and 4 may be done more frequently. Step 1 may be done independently of step 2 also, so that new configurations for step 4 may easily be applied (e.g. changes to the knowledge base). Figure 3.1 shows the mining workflow for SubCat.

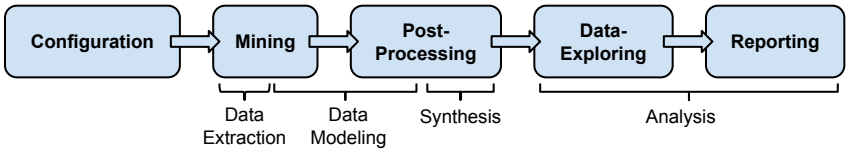


Figure 3.1:
Mining process for SubCat as Presented in [8]

Pre-processing Data from Different Repositories

As stated in chapter 2 by Hassan in [76] and identified as **Challenge 1 and 2** in the previous sections, it is necessary to extend the access of data mining tools beyond single technologies or types of repository. This means that the target architecture for SubCat contains a flexible layer of pre-processing capabilities that can be extended easily. Figure 3.2 shows the repository type hierarchy.

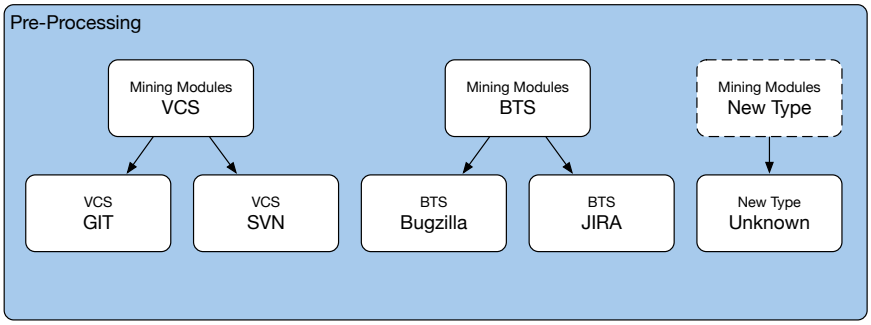


Figure 3.2:
SubCat Architecture - Pre-Processors and Repository Types

Each repository type shares a self-defined meta-model. This means that, e.g. all code repositories map to one generic data model and all issue trackers map to one generic data model. If there would be new types of repositories to be integrated into the SubCat methodology, e.g. mailing lists, all mailing lists would share one generic data model. The idea is that the differences in the specific repository types are transparent for the post-processing basic analysis and NLP components. Each pre-processor must be capable of being triggered individually and no coupling between the pre-processors may exist. By separating the pre-processors from the post-processors it is also possible to split the mining process into two parts, based on their frequency of application, thus reducing overall analysis runtime and thus improving performance of the framework (addressing **Challenge 5**).

Robust Data Model

To provide a robust tool that is resilient to change, it is necessary to conceptualize a data model that is flexible enough to integrate various data sources and to provide this data to different output channels, as proposed by **Challenge 2 and 4**. The target of the SubCat methodology is to gather information about the software engineering process. In detail, the methodology proposes to apply NLP techniques on software life cycle artifacts to extract information. Hence, it is necessary to identify the aspects SubCat analyses. In the first step to design the data model, we define the following entities that are associated with the SubCat methodology:

- Users and identities: research on software evolution centers not only on the delivered software, but also on the people that develop the software. To be able to learn more about the people in projects, it is necessary to know who is who in a project. Since it is possible that a user may use intentionally or accidentally (differently capitalized user name in the repositories) more than one identity in various data sources, e.g. repositories, or chats

and so on, it is important to keep track of users and their identities to produce meaningful reports

- Project to analyze and mine: one of the main design aspects of the SubCat methodology is the transferability of the approach between projects. Hence, SubCat needs to be able to organize and store information on different projects and to keep track of users and analysis artifacts for each project separately

- Bug/Issue from a BTS: one of SubCat focuses is on information that can be gathered from BTS. The key artifact is the single bug/issue that is created, enriched with meta-data and commented in a BTS. Since SubCat does not focus on a single BTS technology, the corresponding entity is generic and applicable to various known BTS. Its current design is based on the Bugzilla issue and is compatible with JIRA, Github Issue Tracker and Redmine

- Commit from a VCS: the other focus of SubCat lies on commits that are entered into a VCS. As is with BTS, the current data model is generic for commits and applicable for at least SVN and GIT without any customization.

- File: an important aspect for software evolution lies on the evolution of code pieces themselves, i.e. the files that are being written in their specific programming language. SubCat needs to know meta-information about the files of a software project, so according reports and analysis data may be provided - e.g. how many bug-fixes happened on the same file.

- Various Analysis Entities: additionally, aside from the entities under analysis, SubCat applies specialized procedures that require data to be stored:

  - Dictionary: a dictionary for classification of bodies of text. In contrast to generic classifiers, dictionaries hold contextualized information, e.g. security relevant terms or terms that allow for classification into software maintenance categories
  - Category: the categories that make up a dictionary. A dictionary consists of one or more categories which are associated with contextualized terms.
  - Interaction: SubCat also tracks interactions between users. This information may be mined from BTS entries and comments, but also mailing lists, chats and VCS
  - Sentiment: Since one of the goals of SubCat is to provide sentiment analysis, sentiments for blocks of texts need to be stored. Hence, sentiment is aggregated for each sentence and paragraph and cumulated for a textual entity (a commit in VCS or a comment in BTS) - both sentence and paragraph sentiment is also stored

Figure 3.3 shows the final design of the data model that shall be transfered into the technical data model.

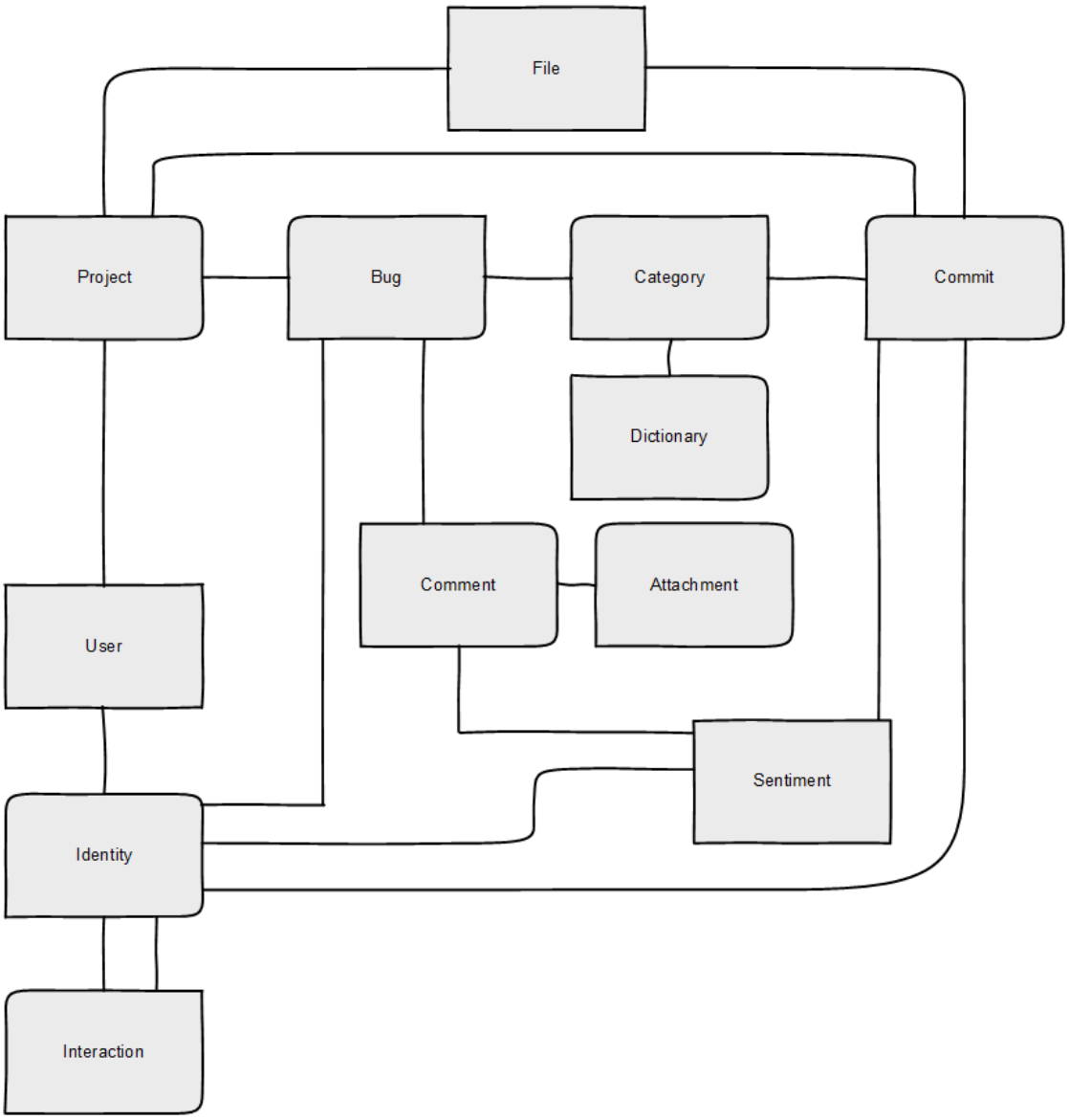


Figure 3.3:
SubCat Data Model - Entities

3.3 Integration of Analysis Tools

The next step after data has been aggregated by the pre-processors is to apply analysis tools that transform the raw data of the pre-processors into valuable information. The SubCat method-

ology focuses on NLP-techniques for this transformation to tackle **Challenge 1**. However, the simple application of NLP-techniques on mined raw data has its limitations and customized NLP-functionality alongside existing libraries should be integrated into the framework. Furthermore, it is important to integrate the various knowledge bases with each other, so while the methodology focuses on NLP-techniques, the framework should also include convenience mining functionality, which may be further used to interpret the NLP analysis results. Figure 3.4 shows the basic separation of the post-processing layer.

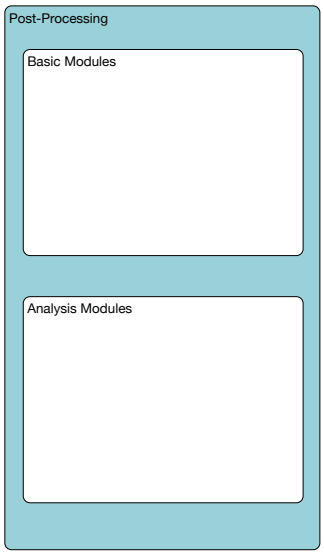


Figure 3.4:
SubCat Architecture - Basic Separation of Post-Processors

NLP tools

Chapter 2 indicates that there are two major limitations in existing approaches in the field of software evolution research that are relevant for the SubCat methodology. The first issue is the lack of relevance of existing approaches in the software development community. While tools, e.g. like Mylyn Reviews [2] origin in the scientific area and consequently have managed their way into commonly used technology, this cannot be said for tools originating in the software evolution research area, which still lead a fringe existence. This is also addressed as a major concern in the field of software evolution research - see [75].

The second issue is closely related to the first issue and one of the reasons of the low acceptance. Approaches in the area of software evolution research are often complicated to set up

[2] `https://projects.eclipse.org/projects/mylyn.reviews`

and difficult to integrate into existing tool chains. Furthermore, they are not robust and deliver results that require expert knowledge for result interpretation. Overall, usability of these tools is very restricted. As argued in the earlier chapters, we believe that insights into software evolution are important to improve software engineering as a discipline over all, but also may be a useful management tool to provide metric-based analysis about the state of a software project.

To tackle the second issue of usability and applicability, we need to find a way to efficiently leverage NLP-techniques for mining software repositories. NLP libraries themselves are a great example for ease of use and applicability, as they are frequently incorporated into various pieces of software. This is possible because NLP uses a homogeneous base, i.e. the English language, and therefore is applicable on any artifact in the English language. As already explained in chapter 2, there are several approaches that exist, when harnessing NLP techniques for software evolution research. These may be split into three areas:

- Application of basic NLP techniques to directly analyze software artifacts
- Application of basic NLP techniques and contextualized classification techniques
- Application of basic NLP techniques and advanced NLP techniques like Sentiment Analysis or machine learners in the context of NLP

The first approach was used earlier in software evolution research and aimed to discover singular findings (e.g. duplicate bug reports). The benefit of this approach is the ease of applicability, since no expert knowledge is required to interpret the results. The major drawback of these types of approaches is that the information gathered seems to have only low value to software engineers. No further application scenarios were analyzed and neither was the usefulness of these approaches qualified. So, while using NLP techniques directly is useful for fundamental knowledge gathering on the software evolution process, applying only NLP techniques on software artifacts does not seem to be overall considered beneficial in the daily software development life.

The second and third approach were established around the same time and face to some extent the same challenges. Both approaches use basic NLP techniques to normalize bodies of texts that are part of software related artifacts and then use more complex mechanics to e.g. provide labels for software artifacts based on the meta-data mined from these software related artifacts. The difference between these two approaches is that the third approach uses advanced techniques provided by generic libraries without proper domain context. The second approach implements domain specific information, e.g. a domain specific dictionary, and uses this information to map natural language artifacts. Some advanced techniques transfer well into the

domain per se, but suffer from artifact granularity (e.g. using Sentiment Analysis on commit messages), while some do not transfer well and are only useful in the context of one project (e.g. machine learners). However, appliances like the combination of machine learners and customized domain specific labeling show promising results. Since SubCat shall be able to provide all this, the architecture needs to allow for all these features.

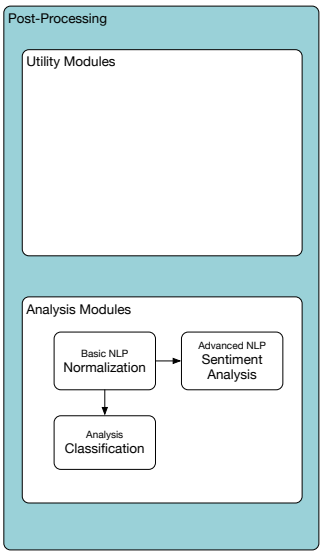


Figure 3.5:
SubCat Architecture - Post-Processors: NLP Modules

Convenient Mining Tools

While the focus of the SubCat methodology is to provide NLP techniques and analysis tools based on them in software evolution research, it is also important to connect various data sources with each other. One step necessary for e.g. conjoint analysis of repositories is to link accounts or artifacts together. To manage this, the post-processors also include a component that includes basic modules for mining convenience, where functionality that transforms raw data into further usable data may be provided. This so-called cross repository functionality has been identified as a need for software evolution research tools in the earlier mentioned studies and has been defined in **Challenges 3 and 4**. This functionality is useful in analysis scenarios where e.g. ID matching for software repositories allows to analyze how developer activity in BTS and VCS corresponds and compare task profiles mined from VCS to expressed sentiments in a BTS. To keep this functionality separated, it is placed in an own module, so separation of concerns and flexibility in the orchestration of different mining steps may be achieved.

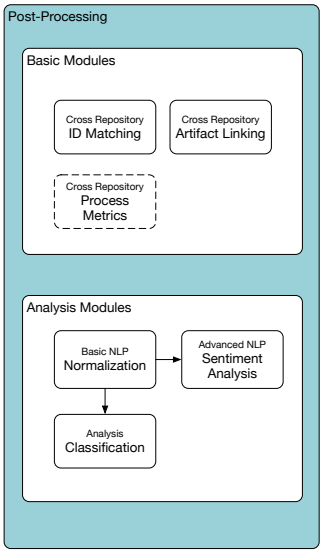


Figure 3.6:
SubCat Architecture - Post-Processors: Basic Modules and NLP Modules

3.4 Presentation Layer

To be able to address **Challenge 3**, SubCat will need a specialized layer for presentation, since various output forms are required for different possible scenarios. The framework shall support research and management reporting, so table-based reports are necessary. These may be provided as comma-separated-value files or as XML. Further, a stand-alone tool offering cockpit-like functionality for project managers and researchers shall be offered. Since the SubCat framework implements many metrics that give an overview of the state of a project, it may also be integrated into a build process as a service and generate JavaScript charts for integration into web-based project dashboards.

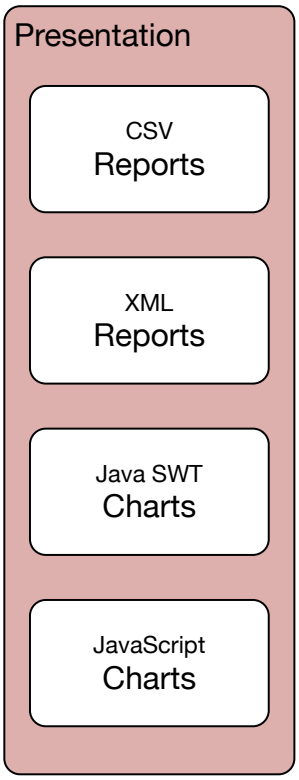


Figure 3.7:
SubCat Architecture - Presentation Layer Overview

3.5 Design of a Framework to Use in Software Evolution Research

By providing means to include domain-specific knowledge, SubCat mitigates one of the previously mentioned drawbacks of existing approaches per design. The SubCat methodology allows a two-tier approach in post-processing. It provides basic NLP functionality, which can be used on data gathered from various repositories in the software development tooling landscape. The second step of the methodology uses the gathered data and applies specialized techniques like domain specific labeling or extended NLP approaches like Sentiment Analysis to provide information on the software evolution process. This information then may be visualized in reports or different tools.

After the three major building blocks are established, the next step is the design of the framework that includes these building blocks. Since this framework should be expendable beyond the knowledge base presented in the previous section, and since this framework should be easily integrated into existing tool chains, certain architectural decisions have to be made to decompose

the framework into building blocks.

As indicated in the previous chapter, mining for soft-factors like motivation and psychology or how open source team members interact with each other has become prominent in software evolution research. This emphasizes the initial assumption that while mining on technical, structured data is important, mining of non-structured data and robust methodologies to analyze this data are needed. A combination of these data sources in fundamental work like Hindle's in [29] that applies NLP on technical artifacts is extremely important. The SubCat methodology has been designed to deliver insights on the intent and the motivation behind the actions of contributors and members of a software development project.

In chapter 2 we provided a literature review on how important social interactions are for long term motivation. However, as other research also suggests (e.g. Godfrey in [52]), it is important to differentiate on this. There is the hobbyist and the need-oriented contributor, there are the core developers and the part-time contributors (compare to Mockus in [3]). There is a study by Hong et al. in [66] that shows the impact of development members on others, which is especially intriguing in context with Lin's work on community patterns [5]. However, this research is not analyzing causal relations, but is a posteriori. A core developer leaves a community and the community breaks up and changes into separate communities. We might derive from a mailing list the last, obvious reason for this, but emotions are built up over time and from continued experiences. SubCat intends to offer data for analyzing the chores of developers in open source projects and to see how their behavior changes over time and see if these connect with emotions by applying sentiment analysis on repository information similar to work by Guzman et al. in [102].

Insights gathered in this fashion could be combined with works by e.g. Bird et al. in [65] where they show that community structure is reflected in development tasks in open source projects. Thus it might be possible to find out who is e.g. a "maintainer" and whether this person is satisfied with this role. Any implementation of the SubCat methodology therefore is required to integrate data sources and mining techniques that are mandatory to deliver this data.

To be able to cater these identified diverse needs, the design of the framework architecture is very important.We used lessons learned in existing approaches for the framework architecture and used the architecture of Kenyon[9], seen in figure 3.8, as a baseline and the formulated challenges as goals. The implementation shall use a DB to store mined information, so other projects/studies may use pre-processing modules (see chapter 10 for an example). The implementation shall use configuration files, so attributes may be changed at runtime, because this has proven useful in our software development experience. The implementation shall use a layered architecture pattern, because pre-processing is done less frequently than post-processing with new attributes and there are transactional stopping points between the various layers. This is

also a consideration of **Challenge 5**, since a split process allows for a certain scalability.

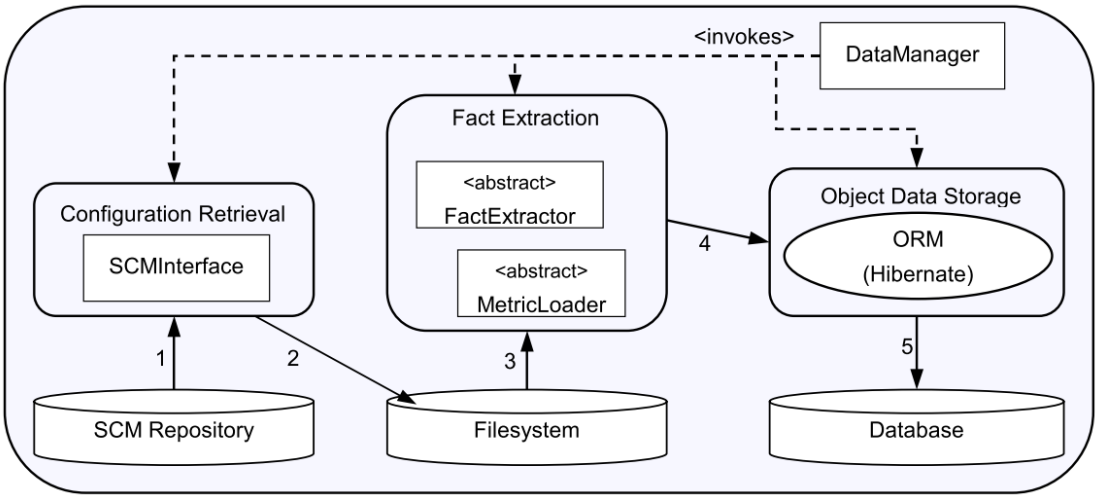


Figure 3.8:
Example of Repository Mining Architecture: Kenyon by Bevan et al. in [9]

Based on the design decisions in the previous sections and the assumptions in this section, the SubCat methodology uses the overall architecture presented in figure 3.9.

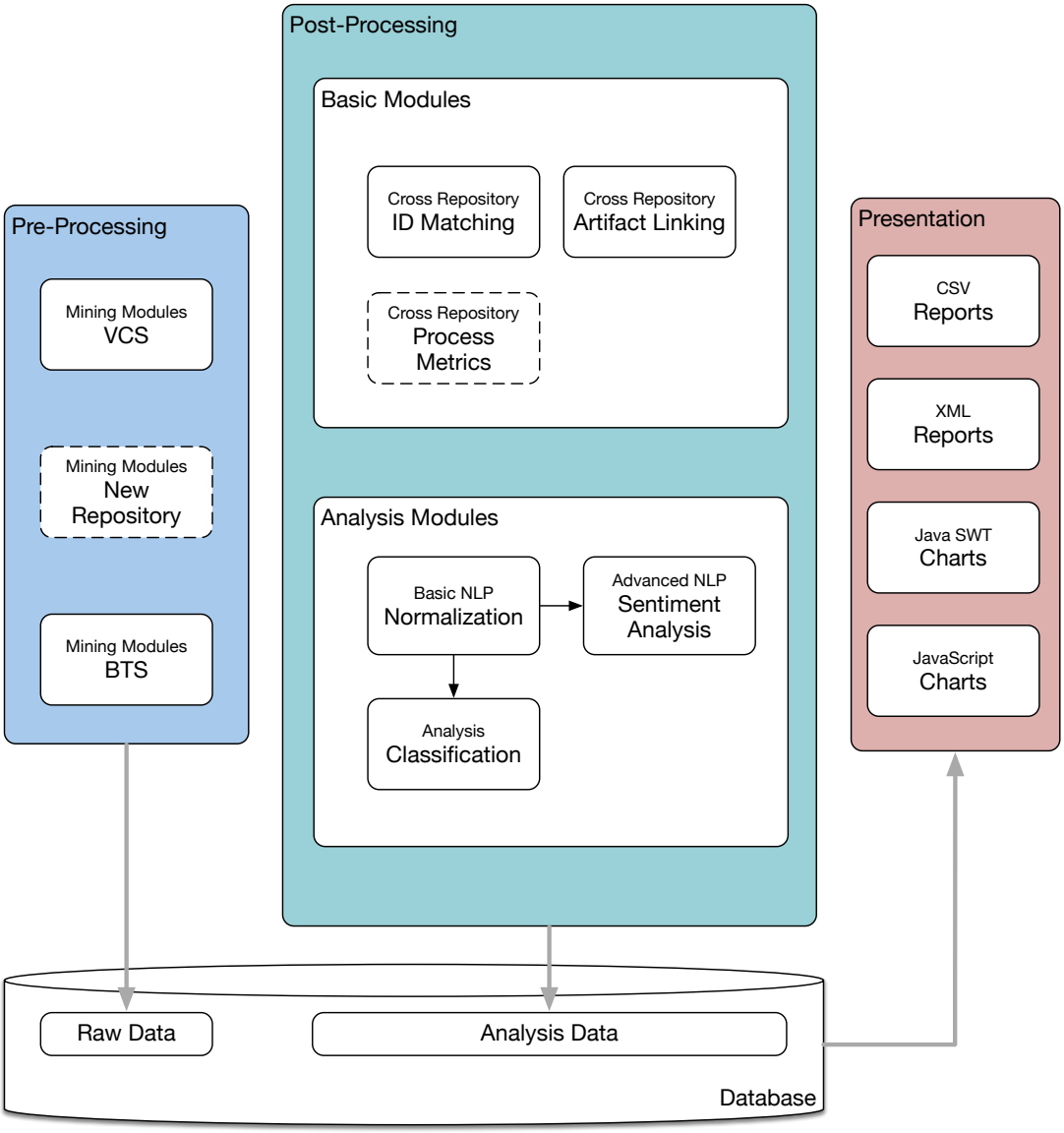


Figure 3.9:
SubCat Architecture - Overview

CHAPTER 4

Implementation of the SubCat Methodology

Contents

Plan to throw one away

The Mythical Man Month [103]

This chapter describes prominent features of the current implementation of SubCat used in the applications of the methodology in part III. It contains functional and non-functional requirements of the implementation. It also explains how a researcher or manager may use this implementation and customize it for targeted analysis by providing usage scenarios. To ease the readability of the following chapter, we refer to the current implementation of the SubCat methodology only as SubCat. The chapter is structured as follows:

- Section **Functional Requirements** lists project characteristics that may be measured by the implementation, it describes in detail the implemented classification algorithm and sentiment analysis. It describes the reports and views provided to the end-user
- Section **Non-Functional Requirements** maps non-functional requirements based on the previously identified challenges to their solution in the actual modularization and im-

plementation of SubCat as well as describes SubCat's current dependencies on external libraries

4.1 Functional Requirements

The functional requirements for SubCat are based on the needs defined by the applications of the SubCat methodology for actual studies and surveys presented in part III. SubCat provides data on project characteristics as listed in 4.1. A complete specification of SubCat's implementation features may be found in [40].

Project Characteristic	**Description**
Number of past faults	Supports identification of problematic modules (used in chapter 5)
Number of changes	Supports identification of problematic modules (used in chapter 5)
Number of authors	Supports identification of problematic modules (used in chapter 5)
Number of categorized changes (maintenance tasks)	Supports profiling of modules and authors (used in chapters 6, 8 and 9)
Number of categorized changes (security tasks)	Detection of security related modules (used in chapter 7)
Number, size of commits	Valuable information about changes for follow-up analysis (used in chapters 6, 7 , 8 and 9)
Overall project sentiment in textual artifacts	Shows the general attitude of a project community in textual artifacts (for future works, see chapter 12)
Extent of collaboration between team members	Recurring collaborations from BTS tickets between team members (for future works, see chapter 12)
Number of interactions that lead to closed issue	Number of comments until issue is closed in BTS (for future works, see chapter 12)
Sentiment rate of interactions	Measure the sentiment/mood of interactions in BTS issues and VCS commits for future works, see chapter 12)

Table 4.1: Excerpt of Project Characteristics Generated by SubCat

Aside from convenience functionality provided in post-processing modules, SubCat includes specific features which are described in the following sections and used in the applications of

SubCat in part III.

Classification Algorithm

SubCat implements a customized classification algorithm that is based on a dictionary and used in the generation and evaluation of a cross-project valid dictionary for task classification in chapter 6. A dictionary is a set of absolute and relative categories defined as follows:

A term in a Category:	ct
Relative Category:	$C_r = \{ct \vert bug, fix, problem, ..\}$
Absolute Category:	$C_a = \{ct \vert cvs2svn, ..\}$
Dictionary:	$D = \{C_{r1}, .., C_{rn-1}, C_{rn}, C_{a1}, .., C_{an-1}, C_{an}\}$

In the specific context of the presented study, SubCat implements Swanson's maintenance categories as well as a blacklist category to filter repository specific transactions (e.g. cvs2svn migrations, branching, etc.). In general, SubCat implements two kinds of categories, of which relative categories may be ranked (e.g. for task classification, categories are ranked based on expected occurrence in the order corrective, adaptive, perfective):

- Absolute Category: This type of category is used for filtering of text bodies that should not be taken into consideration for classification. Keywords that disqualify commits for further analysis are included here, e.g. administrative changes. If one of the terms of this category is found in a commit, it will not be further categorized.
- Relative Category: This type of category is used for regular categorization. Terms in this category are weighted and all normal categories are applied to a text body.

SubCat uses these category types for the following algorithm for classification (a description on the validation of the algorithm may be found in chapter 6):

1. Input: Set of absolute and relative categories Dictionary D,
   set of text bodies $M = \{m_1, m_2, ..., m_{max}\}$
2. Select and remove an unclassified body of text from M
3. EXIT in case of empty set.
4. Build lemma $l_1, l_2..l_n$ for every term $t_1, t_2..t_n$ of body of text
5. For each absolute category C_a in D:
   a) For each lemma l_i in the current body of text
      i. For each lemma ct_j in the absolute category
         A. If $l_i == ct_j$ mark the category as possible match (Break and continue with next absolute category)
6. In case a possible match in one or more categories is found
   a) Pick the matching category with the highest rank in the dictionary
   b) Process next body of text (Go to Step 2)
7. For each relative category C_r in D:
   a) For each lemma l_i in the current body of text
      i. For each lemma ct_j in the relative category
         A. If $l_i == ct_j$ add the weight of ct_j to the category score
8. If there is one or more relative category with a category score > 0
   a) Choose relative category with highest score
   b) Resolve collision by category rank of score is tied
9. Process next body of text (Go to Step 2)

The dictionary is stored in XML format (see listing 4.1 for an example). It is easily customizable and extendable. Different dictionaries may be used by supplying corresponding input parameters via command line.

Listing 4.1: Example of a Dictionary

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary title="WeightedDict" >
        <class name="blacklist" weight="1" absolute="true" >
                <word name="cvs" weight="1" />
                .
                .
                .

        </class>
        <class name="adaptive" weight="2" >
                <word name="better" weight="1"/>
                .
                .
                .

        </class>
        <class name="corrective" weight="1" >
                <word name="fix" weight="2"/>
                .
                .
                .

        </class>
        <class name="perfective" weight="1">
                <word name="dead" weight="2"/>
                .
                .
                .

        </class>
</dictionary>
```

Sentiment Analysis

SubCat uses sentiment analysis to assess communication via comments on issues in a BTS and commit messages in the VCS (similar to the works of Guzman in [102]) in chapter 9. The overall sentiment analysis is a two-step process that has a syntactical phase and a semantic phase. Additionally, to extract sentiments in the BTS, SubCat extracts a graph describing the communication between users, which is then used to perform a sentiment analysis for each comment and between users.

- In the syntactical phase, abstract syntax trees are extracted. Before this extraction each comment is normalized. This normalization is achieved by using regular expressions which may be extended to support filtering of technical structures like runtime errors, stack traces, etc.
- SubCat uses the Visitor Pattern [104] to implement the semantic phase. Goals of this phase are to calculate sentiment per comment and between author of comment and addressed persons.

SubCat employs several heuristics to mine sentiments in the BTS during the semantical phase. SubCat uses message flows over comments to assign comments to other users (e.g. if two persons comment on an issue, SubCat assumes they have a conversation). If there are more than two persons involved, SubCat tries to assign sentiment between involved persons based on paragraphs. SubCat also uses quotes to find out who is addressing whom. Closer comments are searched first and the first finding of the quoted text is used. Contents of paragraphs are only analyzed if the other steps fail to find any connection between the commentators. SubCat searches for person names in paragraphs and matches these to the users who contributed to the issue and on locally relevant account names using a Burkhard-Keller Tree [105] (SubCat uses this also for account interlinking between repositories).

SubCat calculates sentiment per paragraph and ignores quotes. We use the Stanford Sentiment Treebank to classify sentences based on the following categories:

s_0 : negative
s_1 : somewhat negative
s_2 : neutral
s_3 : somewhat positive
s_4 : positive

For each sentence a matrix of sentiments $S^T = [s_0\ s_1\ s_2\ s_3\ s_4]$ is provided where:

$$0 \geq s_{0,1,2,3,4} \leq 1$$

The sentiment for the sentence is the one with the highest score. SubCat stores these sentiments per sentence level together with their word counts and aggregates sentiment over paragraphs.We also store optionally the identity the block is addressed to. Using this implementation without any data pruning/filtering resulted in very poor performance. This bad performance was caused by a small set of comments, which included technical information blocks like stack traces - see listing 4.2 for a (shortened) example. After measuring performance times and identifying problematic artifacts, a filter for paragraphs with more than 10 newlines was implemented (for details on the analysis see [40]).

Listing 4.2: Example of a Comment Including a Stack Trace[1]

```
I have tested it and it seems to have some problems, i.e. I can
   ↪  make it crashes.
How to reproduce it:
1. Run gedit
2. Enable View->Highlight Mode->Markup->XML
3. Write <aaa></
4. It closes </aaa>
5. Write <bbb></
6. It closes </bbb>
7. Write <b
8. It crashes

I'm using CVS HEAD for gedit, gtksourceview and gtk+.
Here the stacktrace:

Backtrace was generated from '/opt/gnome/gnome210/INSTALL/bin/
   ↪ gedit'

Using host libthread_db library "/lib/tls/libthread_db.so.1".
[Thread debugging using libthread_db enabled]
[New Thread -151150464 (LWP 29491)]
0x0077b7a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
#0  0x0077b7a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
#1  0x009f80b3 in __waitpid_nocancel () from /lib/tls/
   ↪ libpthread.so.0
#2  0x0039ddda in libgnomeui_segv_handle (signum=11) at gnome-
   ↪ ui-init.c:741
#3  <signal handler called>
#4  0x00c49979 in IA__g_object_remove_weak_pointer (object=0
   ↪ x3667c8,
weak_pointer_location=0x942b6d0) at gobject.c:1549
...
```

[1]Original comment: https://bugzilla.gnome.org/show_bug.cgi?id=163014

To filter potentially large chunks of technical information, SubCat implements a content based paragraph filter that uses the following rules, which further greatly improved performance of sentiment analysis. The filter looks as follows:

- Spaces or tabs at the beginning of lines
- Lines that start with "`make[`".
- Lines that end with "`\`".
- Lines that start with one of the following patterns:

```
\[[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+ (~|[a-zA-Z0-9_-]+)\](\$|#)
[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+:(~|[a-zA-Z0-0_-]+)(\$|#)
```

If one of these listed filter criteria is present, SubCat ignores the paragraph. There is an exception in place for list elements in paragraphs. If SubCat identifies a list, it is treated as ordinary text. After adding these filter criteria, performance was further increased by about a third.

Data Explorer and Reports

The data explorer provides an SWT-based user interface and is used as part of the initial project screening for the applications of SubCat in part III. To support high configurability (*NF03.2*, see next section), SubCat provides a general purpose controller. This controller takes widget specific configuration objects (the configured options and a description of the data to be retrieved). The different types of widgets are a *PieChart diagram*, *Trend diagram*, *Distribution diagram* and *Relationship diagram*. These widgets may be aggregated in three different types of views (Project View, User View, Collaboration View). The main objective of this design approach is to be able to dynamically present new data and to loosely couple the data explorer to the model, so an end user has the possibility to switch to new user interfaces or to integrate SubCat into other data processing tools. How to use the data explorer and the reports and how to configure SubCat is described in the next section.

In addition to the data explorer, SubCat also generates reports in CSV format on:

- Authors
- Changes
- Modules
- Distribution of terms

This number of reports may easily be extended by declaring a specific report in SubCat configuration that includes a corresponding SQL-type query. Reports may also be generated in XML.

Scenarios for Using SubCat

This section focuses on the end user functionality provided by SubCat. For this showcase, SubCat has been configured to show three data exploration views for a project. Under the tab *Pie Charts* SubCat holds pie charts for the commit categories for the whole project and for the rate of commits that are linked to the corresponding BTS. See figure 4.1 for the two charts.

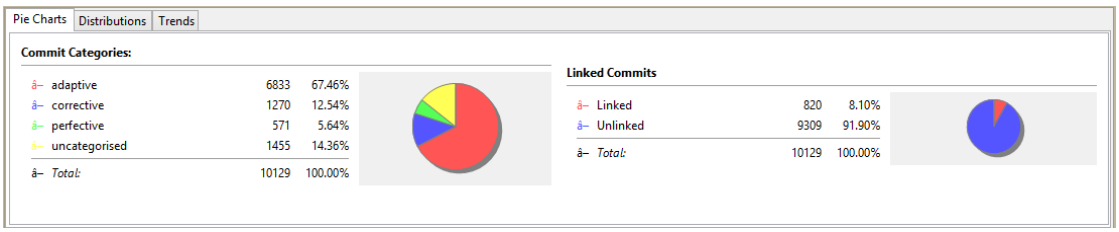


Figure 4.1:
SubCat Examples - Pie Charts Showing Classified Commits and Linked Artifacts

In the distributions tab, SubCat shows box plots and bar charts of the selected project. The sample demonstrates distribution charts for comments per bug that were categorized as adaptive. The input data for the charts is defined solely in the configuration file and is generic - it is e.g. possible to switch the content of the drop down buttons and change the data sources for all these components. See figure 4.3 for the box plot and figure 4.2 for the bar chart.

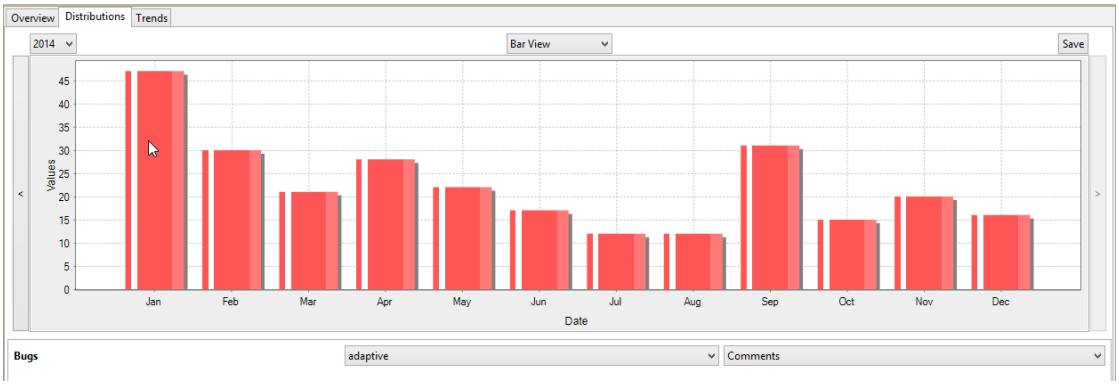


Figure 4.2:
SubCat Examples - Bar Charts Showing Comments Classified as Adaptive

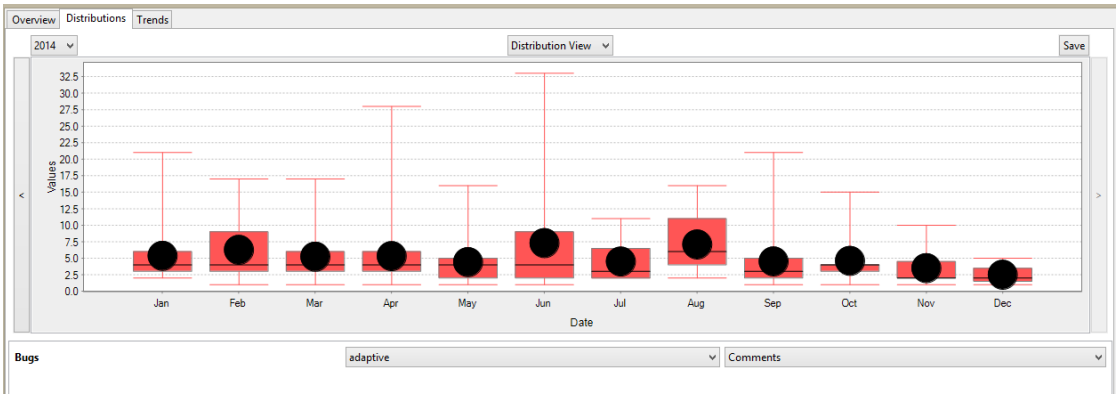


Figure 4.3:
SubCat Examples - Box Plot Showing Comments Classified as Adaptive

In the final view, an example of SubCats developer profile based on classified source changes can be seen. SubCat uses the change date to plot how the profile based on the tasks the developer shifts over time. See figure 4.4 for the trend chart.

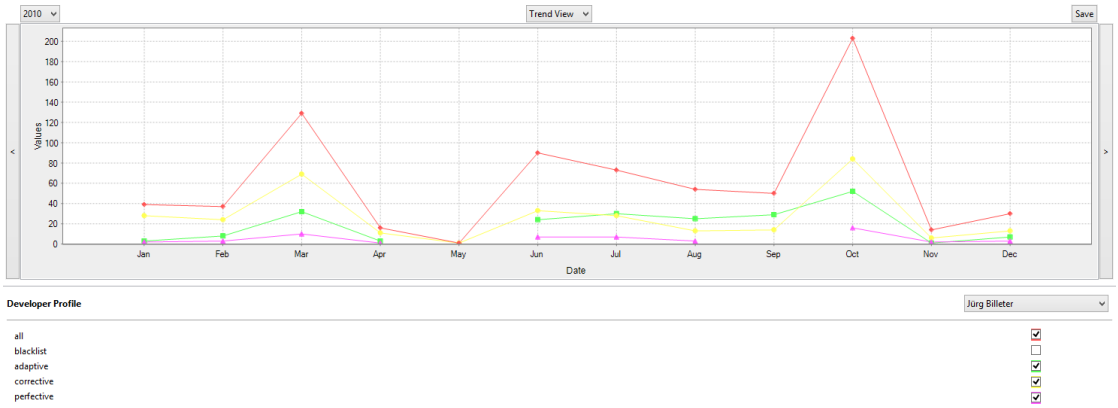


Figure 4.4:
SubCat Examples - Trend Chart for Specific Developer and Maintenance Tasks

Reporting for BTS and VCS

SubCat provides a command line interface (CLI) (see table 4.2) to parametrize the report generator module. Different output types as well as any configured report type may be generated by SubCat.

Parameter	Description
-b,--bug-dictionary <arg>	ID of the bug dictionary to use
-c,--commit-dictionary <arg>	ID of the commit dictionary to use
-C,--config <arg>	Path to the configuration file that includes the reports
-d,--db <arg>	The database to process (required)
-f,--format <arg>	Desired output format of the reports
-F,--list-formats <arg>	List all supported formats
-h,--help	Show command line options
-o,--output <arg>	Target path and file for output
-P,--list-projects	List all registered projects
-p,--project <arg>	The project ID to process
-r,--report <arg>	Type of report
-R,--list-reports <arg>	List all types of report

Table 4.2: Parameters and Descriptions for the Reporter Module

Since reports are generated by providing SQL-like queries, all data from the database may be queried. In the provided small sample, SubCat dumps the content of the categorized commit table into a CSV file.

Table 4.3 shows an excerpt of the report SubCat generated based on the sample configuration in listing 4.3).

Listing 4.3: Sample Report Configuration

```
Reporter = {
Name = "Commits dump";
Query = "SELECT * From Commits";
};
```

Id	Identifier	Author ID	Committer ID	Category
1	57b0a0f1d3fae0a34...	5926	5926	adaptive
2	608d7a6b9b7f6d713...	5926	5926	adaptive
3	801d3852a339a5b0a...	5926	5926	adaptive
4	2fb5b83530f635f68...	5926	5926	corrective
5	c23fdcd236fba4fa8...	5926	5926	corrective

Table 4.3: Resulting Report for the Reporter Module

4.2 Non-Functional Requirements

The identified challenges in chapter 3 have been translated into non-functional requirements (NFR) and listed in table 4.4. NFR are then mapped to the internal structure of SubCat.

NF01: Demeyer [75] - Pre-processors (i.e. miners) shall be extend-able for new miner types	
NF01.1	VCS miners shall include SVN and GIT
NF01.2	BTS miner shall at least include Bugzilla
NF01.3	VCS miners may easily be extended for other VCS, e.g. Mercurial
NF01.4	BTS miners may easily be extended for other BTS, e.g. JIRA
NF02: Hassan [76] - MSR-techniques shall be simple and convenient to use for untrained users	
NF02.1	SubCat shall provide scalability and be implemented with high performance in mind
NF02.2	SubCat shall provide user-friendly interfaces and reports usably by practitioners as well as researchers
NF03: Additional requirements	
NF03.1	SubCat shall integrate more than one repository type
NF03.2	SubCat shall be highly configurable

Table 4.4: Non-Functional Requirements of SubCat

Figure 4.5 shows the internal structuring of SubCat. This structure covers most of the non-functional requirements listed in the previous section. The purple arrows show the mining work flow, green arrows indicate data flows. *Settings* encapsulate the submitted parameters required for usage of the functionality offered by SubCat, while *Config* consists of configuration items for the *Data Explorer* and *Reporter* capabilities (e.g. customized data queries) to visualize the aggregated data. The *Model* is used to persist data in the database.

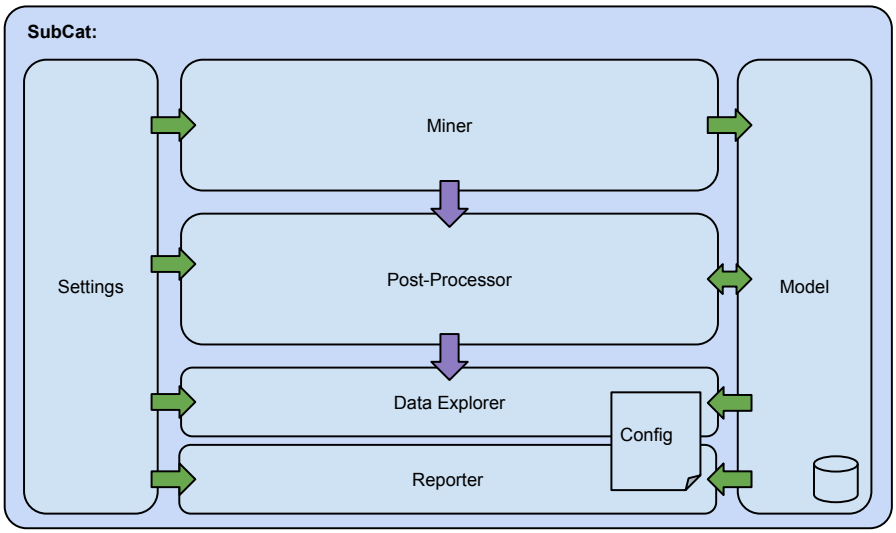


Figure 4.5:
Architectural Overview for SubCat as Presented in [40]

Pre-processors/Miners

SubCat pre-processes data by applying different miners on repositories. These miners may be differentiated by repository type (BTS or VCS - *NF03.1*) and system (SVN or GIT - *NF01.1*). These miners are running in threads and may be multi-threaded where applicable (e.g. in Bugzilla and GIT) and single-threaded where necessary (e.g. SVN because of non-atomar file history). This design ensures scalability and performance were applicable (*NF02.1*). Miners may be extended by creating new subclasses of the existing miner infrastructure (*NF01.3* and *NF01.4*). This means that a new miner for a BTS repository like JIRA may easily be implemented as long as the miner handles its data within the boundaries of the SubCat model.

Post-processors

After pre-processing by the various miners, post-processors are used to generate the data on project characteristics (see table 4.1). The main function of post-processors is to prune and transform the raw data from the miners to data that may be used in reports and analysis. Post-processors may also be run in threads (*NF02.1*). Post-processors currently include the following functionality:

- User matching in VCS and BTS based on heuristics
- Comment and bug interlinking

- Classification of commit messages
- Classification of BTS issues
- Sentiment Analysis

A detailed technical description of the current implementation may be found in [40]

Dependencies

Since SubCat is open source software, we use only open source libraries with compatible license models. A list of technical dependencies of SubCat can be seen in table 4.5 - SubCat uses Maven to manage these dependencies.

Library	Description
General	
Java[2]	Programming language of the tool
SQLite[3]	RDBMS for data storage
JDBC	Data Access API
BTS Mining	
Apache RPC-XML[4]	To access Bugzilla API[5]
VCS Mining	
JGit[6]	Java Library to work with GIT
Text Mining	
Stanford NLP[7]	Library used for lemma building
SentiWordNet[8]	Library used for sentiment analysis
CLI/Data Explorer/Viewer	
Apache Commons CLI[9]	Library for command line input parsing
SWT[10]	Graphical front end
JFreeChart[11]	Library to display data in charts
Misc.	
Maven[12]	For dependency management

Table 4.5: Dependencies of SubCat

[2] http://www.java.com/en/
[3] http://www.sqlite.org/
[4] https://ws.apache.org/xmlrpc
[5] http://www.bugzilla.org/docs/3.6/en/html/api/Bugzilla/WebService/Server/XMLRPC.html
[6] https://eclipse.org/jgit/
[7] http://nlp.stanford.edu/
[8] http://sentiwordnet.isti.cnr.it/
[9] http://commons.apache.org/proper/commons-cli/
[10] https://www.eclipse.org/swt/
[11] http://www.jfree.org/jfreechart/
[12] http://maven.apache.org/

Part III

Application

In this part of the thesis, we show the various applications of SubCat as a mining tool to prove the validity and usefulness of the proposed methodology. Several of the following chapters have been presented at notable peer-reviewed conferences and have been published accordingly. In the presented applications of SubCat we show how the methodology may be implemented and applied by practitioners to populate a bug database from scratch or how SubCat may be used by security researchers to find security relevant modules. We also show how SubCat may be used by managers to learn about social interactions in their development team and which sentiments are expressed in the project. Further, SubCat is used to gather common metrics and to create and evaluate a dictionary for change classification that is valid for the open source domain. The chapter is structured as follows:

- Section **Applying the SubCat Methodology for a Preliminary Feasability Study** shows how the SubCat methodology was implemented to use a dictionary based on existing literature for an analysis of the relationship of process metrics and classified changes
- Section **Applying the SubCat Methodology for Security Analysis** shows how the methodology is implemented using a specific, security focused dictionary to classify security critical components
- Section **Applying the SubCat Methodology to Populate an Issue Tracker** describes how the methodology was implemented to extract maintenance information out of repositories. This information is used to automatically generate issue tracker entries from an existing repository
- Section **Applying the SubCat Methodology to Create Developer Profiles** shows how SubCat may be used to generate developer profiles of open source projects by using maintenance task classification and sentiment analysis
- Section **Applying the SubCat Methodology in a Survey for Commit Classification** shows how the overall mining framework, which comes as part of the implementation of the SubCat methodology was used to mine open source projects and to provide data exports for surveys and related studies

CHAPTER 5

Applying the SubCat Methodology for a Preliminary Feasability Study

Contents

The following section was partly published by Mauczka et al. in [10] at the SEKE conference to showcase the implications of the SubCat methodology on the analysis of the relationship of classified change data and process metrics.

5.1 Introduction

The preliminary study was conducted to learn whether the approach of using NLP techniques for software evolution research was feasible in itself, i.e. that classification from prior work could be reenacted, and how this information might be used in concurrence with other mined project data to e.g. identify problematic modules. Further, the preliminary study was conducted to validate design and architectural decisions for the SubCat methodology. In the preliminary study, we did not redefine Swanson's categories as described in chapter 2, so the definition by Mockus in [37] was used - corrective changes are changes to fix faults, adaptive changes are

changes that add new features and perfective changes are changes that restructure the code to accommodate future changes.

5.2 Presenting the Idea and the Data

Predicting fault data as shown by Zimmermann et al. in [106] is a helpful source to learn more about the nature of software projects. However, from a project management point of view, finding or predicting bugs is only one part of improving software quality. To prevent bugs and to keep the number of bugs low is as much of a priority as knowledge about possible bugs. Performance increase and keeping the system scalable, stable and secure is important as well. These problems are not addressed by analyzing or predicting bugs alone. To analyze most of these factors, we will examine the relationship of process metrics and corrective, adaptive and perfective changes.

Classifying Change

We classify changes using an approach introduced by Mockus et al. in [37]. To test our approach we chose the Core Module of the Ant Project. We use scripts based on previous work of Mockus et al.[1] - a stand-alone tool that integrates further process metrics and gathers the project data (bug database and version control repository data) was implemented after the preliminary analysis (see part II for a detailed description of the implementation). The change data is gathered from a VCS, in this case it is Subversion (SVN)[2], though the methodology has been proven to work for CVS as well (see [37]). By using SVN instead of CVS, there is no need to group changes into Modification Requests (MRs) - committing of multiple files is possible and each commit to the SVN therefore is an MR. A MR consists of "all delta that share login, comment and are recorded within a single three-minute interval" (see [37]). We gathered the changes over different time periods (2008 for Number of Deltas, 2003-2008 for Number of Faults and Developers) for a more diverse view on the data. We use the comments of each MR to classify the changes into corrective, adaptive and perfective changes. Before the classification the comments are normalized as described in [37]. Following the keyword clustering procedure, the classification rules had to be altered to provide meaningful results.

There are different approaches for classifying changes, e.g., the approach presented by Sliwerski et al. in [20]. However we were more interested in the maintenance tasks themselves than in the outcome (i.e., bug introduction). Additionally, we were interested in all changes not

[1] http://mockus.us/oss/

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

just the ones linked to the bug tracking system, as the data from the preliminary study suggests that there are more corrective changes unrelated to bugs in the bug tracking system (see section 8 for details on this particular topic).

Gathering the Process Metrics

The goal of the preliminary study is to prove the feasibility of using NLP techniques as part of the SubCat methodology, so the focus of the study is on change classification - hence for reason of work efficiency a subset of the process metrics used by Graves et al. in [107] was chosen. To get the data from Bugzilla[3] we employed scripts used by Mockus et al. in [108].

The following process metrics are included in the preliminary study:

Number of past faults: Graves et al. found that the number of faults in a module in the future is related to the faults found in the module in a past period of time; future faults are a constant multiple of past faults according to them. By classifying change we may show that the fault rate might actually decrease after a perfective change took place. We gather this metric by mining data from Bugzilla.

Number of deltas: We will use the number of deltas to a module to find trends of increase and decrease of faults depending on the kind of change that has been undertaken. According to Graves et al. a high number of deltas is an indicator for future faults in the module and thus corrective changes. The aim of this analysis is whether there is something like the decay of a perfective change. The data required for this metric is the sum of all changes in a given time period.

Number of developers who have made deltas on the module: We will analyze if the number of developers relates to changes undertaken in the module. A natural assumption here is that a high number of developers might lead to an increase in perfective changes later on in the module. This metric is gathered from the SVN by using StatSVN[4].

5.3 Analyzing the Results

We plan to analyze process metrics and their relationship to changes - one kind of analysis is done by correlating both measures with each other. We use the introduced set of process metrics on each kind of change and on all changes. Since the preliminary study is done with the scope

[3] https://issues.apache.org/bugzilla
[4] http://www.statsvn.org/

of only pointing out possible future directions of research, no correlation analysis was done yet, since only one project was examined.

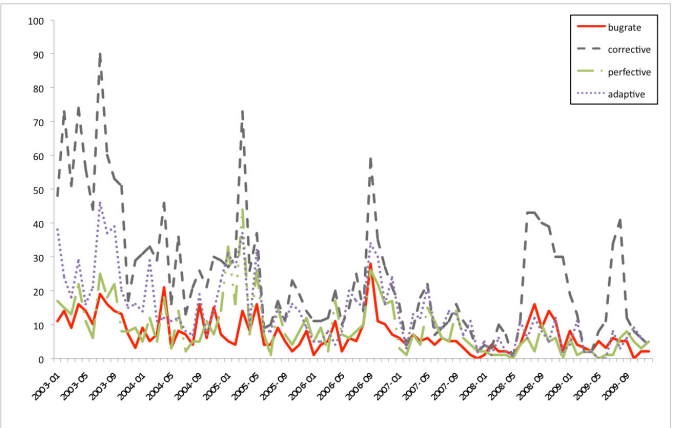


Figure 5.1: Number of Faults and Classified Changes (2003-2008) by Mauczka et al. in [10]

Studies similar to the work presented by Sliwerski et al. in [20] show that there has been some effort to link corrective changes and bugs in a bug tracking system. However, as can be seen in figure 5.1 corrective changes seem like multiples of the bug rate (a bug is not fixed by a single change and a bug might be reopened) - which is expected, but still requires more in-depth analysis (see chapter 8). If a large number of bugs are fixed without ever entering the bug tracking system, predicting fault data by using former fault rates needs to be reconsidered, even if one assumes that the behavior of not reported bugs is a constant factor (i.e., the number of bugs not reported does not change over time). While our preliminary study seems to carry the current assumptions from bug data analysis, perfective changes and their impact on the fault rate provide a challenge for future analysis. When looking at figure 5.1 we see two perfective spikes, which are followed by a decline in the bug rate - however adaptive changes and perfective changes spike there too, so a general change activity is more likely to have occurred than a suggested relationship of perfective changes and a temporarily lowered bug rate.

In figure 5.2 changes are split into the three categories, unspecified changes are not depicted, because the categorization technique itself had not been evaluated at this time (see chapter 6 for an evaluation of a classification dictionary). For the preliminary study we assume the unspecified changes to be distributed according to their categorized appearance (adaptive and corrective accounting for the majority of the changes). About 25% of the changes could not be categorized by our solely automated categorization. We hope to decrease the amount by adding additional features to the mechanism. Figure 5.2 shows summed up changes over the course of a year (2008). While corrective changes seem constant (in accordance to our hypotheses), perfective

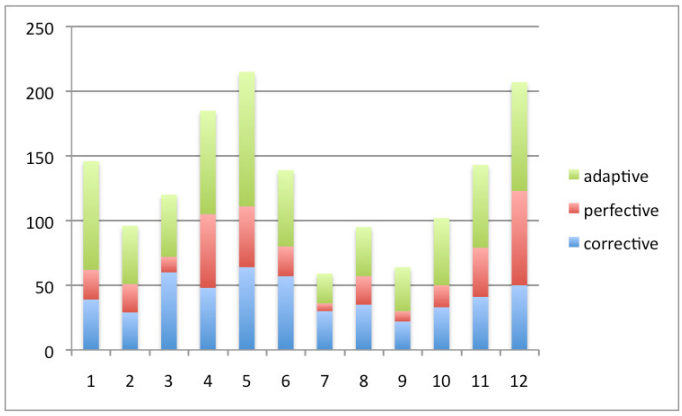


Figure 5.2: Number of Deltas - Classified by Change Category (2008) by Mauczka et al. in [10]

changes peak twice. One of the peaks happens before a peak of corrective and adaptive changes, however the sample size is too small to make any assumptions yet, since it can be seen in figure 5.1 that corrective changes spike over a longer time period and this is in contrast to our hypotheses. An additional process metric to be taken into consideration is feature requests per month, to see whether adaptive changes are demand-driven, or just prone to activity schedules (see figure 5.1 which shows corrective and adaptive changes peaking in similar timeframes). By normalizing the change numbers by activity, we might be able to smoothen the change curves.

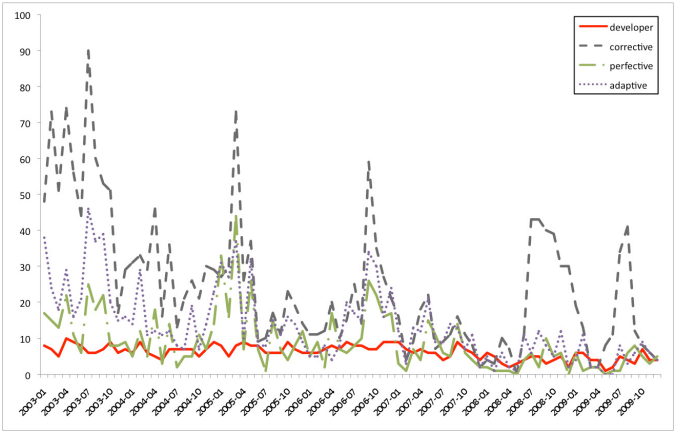


Figure 5.3: Number of Developers and Classified Changes (2003-2008) by Mauczka et al. in [10]

As can be seen in figure 5.3, the number of developers committing to the module seem constant, while the changes show peaks. It can be noted that the developer peak and the change peak fall within the same time frame, however further change peaks are not accompanied by

developer peaks. Even though there are no trends that can be learned from this diagram, the process metric is the number of developers of many modules, so the data gathered for the preliminary study is not sufficient to make any assumptions about how the number of developers and changes of modules relate (past research by Graves et al. show that fault data and number of developers do not relate significantly).

5.4 Outlook

As this is a preliminary study, the threats to validity of the presented study are many. We understand that the change classification approach needs further validation. Our lexical approach is not as successful yet as attempts in previous studies, however we did employ it on different Apache modules with similar performances (about half of the changes classified before tuning it). The follow-up dictionary evaluation study is presented in chapter 6. The preliminary study presented here showcases an example of an analysis that uses an implementation of the SubCat methodology.

5.5 Conclusion

We showed that process metrics and classified changes using the SubCat NLP-based methodology raise interesting questions to expand on in future work. We explained the tools and techniques that may be used in further studies and showed intermediate results on a single module of the Ant Project. We gave insight into the gathered data by showing the development of the metrics and dependent variables over time. One of the most interesting questions is on the relationship of bug tracking and corrective changes - for the field of bug prediction, findings here might have a considerable impact. The experiences made in the preliminary study were used to create the design and the architecture of the SubCat methodology presented in part II.

CHAPTER 6

Applying the SubCat Methodology for Change Classification

Contents

The following section was partly published by Mauczka et al. in [11] at the FASE conference to showcase the application of the SubCat methodology on providing a dictionary for change classification that is transferable between projects and evaluated for the open source domain.

6.1 Introduction

The following chapter describes an implementation of the SubCat methodology for a study to provide a dictionary based classification mechanism, which is then evaluated for cross-project validity. We use meta data which can be mined from VCS that uses commit messages to accompany any change (a commit) to the code base. From the textual information in these commit messages, we mine information about the software evolution process. We base our work on the assumption that commit messages hold information that should give evidence of the purpose of the source code change.

For the sake of readability, the implementation of the SubCat methodology shall be referred to as SubCat for the remainder of this chapter.

SubCat provides different kinds of reports (categorization per file or per module) to visualize the results of this categorization. SubCat also generates statistics on the authors of the commit messages or statistics on the words used in the commit messages.

We used the reports generated by SubCat to create an optimized and cross-project valid dictionary that allowed us to automatically classify commits into Swanson's maintenance categories [12]. To achieve this, we defined an algorithm to incrementally train and improve this dictionary with certain keywords. After training the dictionary on a number of projects, we evaluated this dictionary against a larger set of open source projects.

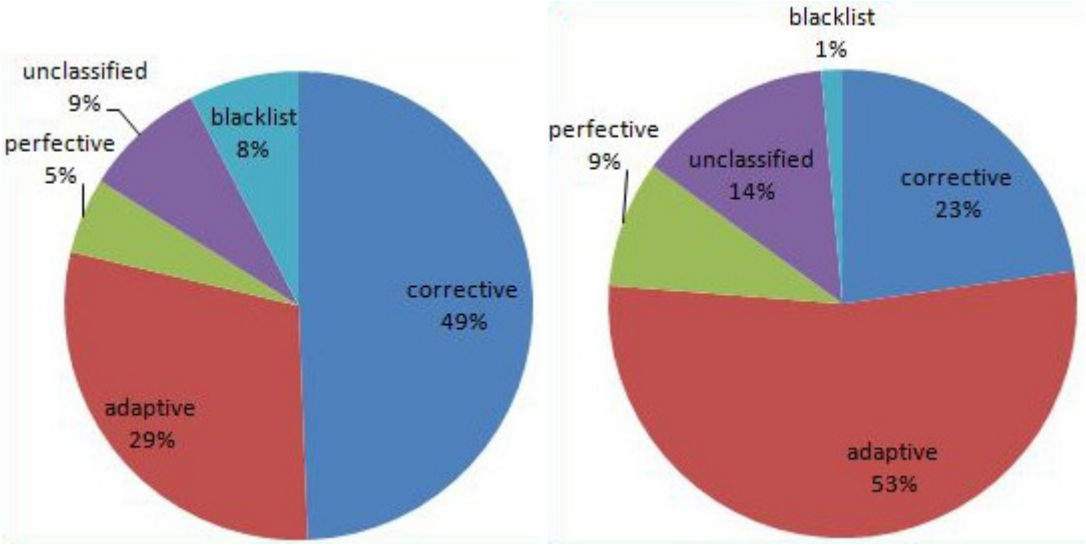


Figure 6.1: Visualization of Classified Activities in Different Software Modules by Mauczka et al. in [11]

By using a corrective classification mechanism, we are able to track bugs within the repository additionally to a normal bug tracker (see chapter 8). This allows for research on the difference of bug granularity in repositories and bug trackers like Bugzilla (a bug fix in the repository may not correspond to a bug report in the tracker). Additionally, we can analyze how developer profiles change over time in a project (e.g. a developer starts in a project to fix bugs that annoy him and ends up implementing a whole new feature - see figure 6.2 for an example). The implementation provides this information, which can be combined with mailing list analysis. This can provide a whole new insight into how a developers profile changes over time in an open-source project.

6.2 Automated Classification Approach

A dictionary, as used in the context of this study, is a set of categories. A category is a group of keywords that share a common meaning and therefore are indicators for this category, e.g. the word "fix" is a keyword for the maintenance category "corrective".

We propose the following procedure to create a dictionary:

Pre-Processing the Meta Information The meta information for our analysis was derived from the commit messages in the VCS. As these messages are written in natural language, we have to normalize them to be able to extract sensible information (e.g. we want to match "this **fixes** a re-ocurring crash" and "I **fixed** an overflow" to its lemma "fix" (a head word under which the word would be found in a dictionary). SubCat uses functionality provided by the NLP-library WordNet[1] to lemmatize commit messages

Initializing the Dictionary We generate an initial seed for a dictionary by referring to prior work (Mockus and Votta in [37] and Hassan in [19]). This initial seed only contains words that hold a high likelihood of indicating a maintenance category

Training the Dictionary To be able to categorize as many changes as possible with a high accuracy for a single project, we use a self-devised algorithm to train the dictionary. We

[1] http://wordnet.princeton.edu/

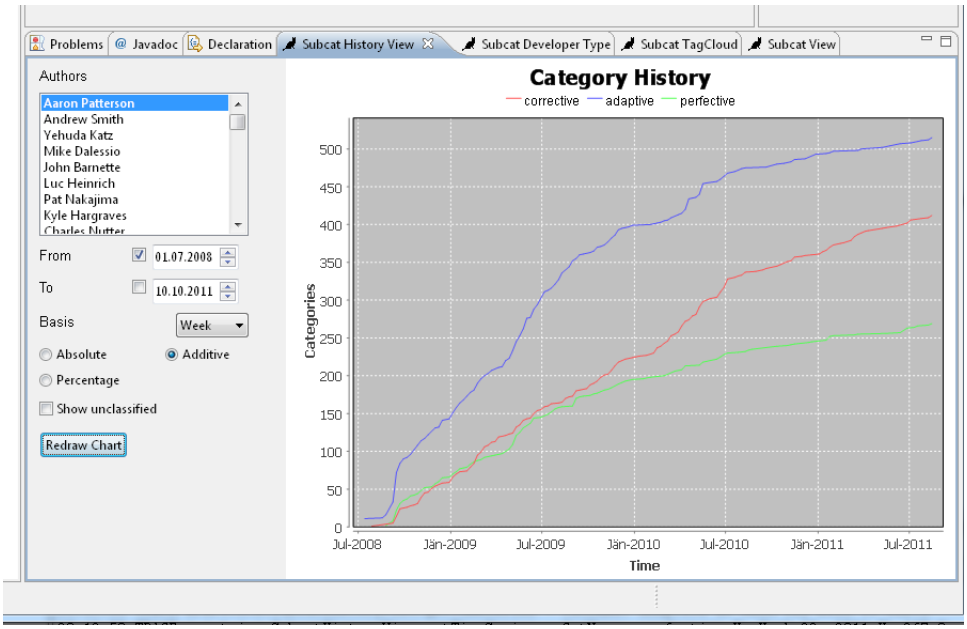


Figure 6.2: Developer Profile by Mauczka et al. in [11]

employ the algorithm to train the dictionary on additional open source projects to further increase the accuracy of the dictionary

Evaluating the Dictionary After the initial training, we use the dictionary on another set of projects to evaluate cross-project validity. We do not further change the dictionary during this step. Only blacklist items (keywords that filter out administrative changes) are introduced

Classification Rules

The research area of the identification and classification of maintenance tasks in the software development process has evolved for decades. In [12], Swanson defines a maintenance task as an activity that can be assigned to one of the following three categories:

Corrective Software Maintenance Activities that are necessary to fix processing failures, performance failures or implementation failures

Adaptive Software Maintenance Activities that focus on changes in the data environment or changes in the processing environment

Perfective Software Maintenance Activities that strive to decrease processing inefficiency, enhance the performance or increase the maintainability

For the development of the automated classification in this study, Swanson's original definition of maintenance tasks is used and slightly extended. An additional category, the "blacklist" is introduced. We use the blacklist category to filter all commits, which underlying modifications were not carried out by humans or which do not actually include any source code modifications. For example commits generated by the "cvs2svn" repository-converter[2] or commits that just "tag" a version. In addition we merged the implementation category, as presented by Hindle et al.[109] with Swanson's adaptive maintenance category. As a result we are able to map every commit to exactly one category. Using Swanson's original maintenance classification provides a categorization into a few, well defined categories and is therefore a suitable starting point to develop an automated classification algorithm.

As mentioned above, our algorithm relies on two sources of information to carry out the classification, namely the commit message and the dictionary. The **commit message** is attached to every commit and encapsulates the information about the intention of the modification. The

[2] `http://cvs2svn.tigris.org/`

dictionary defines the knowledge base for the classification including the categories. The different categories are defined by a set of keywords that indicate that a commit message may belong to this category. In addition, every word has an associated weight (see section 6.3 for details how the weight is generated). The weight value constitutes how strong the indication is. The same word can be contained in multiple categories. See figure 6.3 for a sample dictionary that is used to classify a commit message.

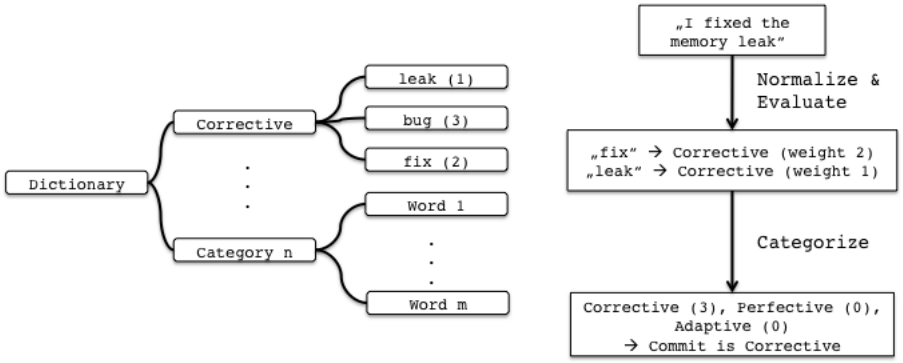


Figure 6.3: Example for Dictionary and Classification by Mauczka et al. in [11]

To implement the blacklist feature, "absolute categories" have been introduced. If a commit message contains a word (e.g. "cvs2svn") that is included in the listing of an absolute category, the commit is instantly assigned to this category, ignoring the weighting mechanism and the normal categories.

Categorization Functionality

For this study, one part of the SubCat methodology was implemented, namely the customizable categorization functionality (see chapter 4). for the study at hand, functionality to mine Subversion was implemented. The results of the classification are reports in the CSV-format. The implementation offers the following reports:

Categorization-Report The categorization report contains all commits and their corresponding classification results in detail. It is the base for the detail reports that follow. It can be used by analysts to generate their own statistics based on the report data. The displayed information per row are: commit including the revision, the category it has been assigned to, the author, the date of the change, the length of its commit message, the overall number

of added and deleted lines for the commit, the score of the commit for each category from the dictionary, the affected modules, the affected files and the revised commit message.

Author-Report The author report shows the analysis of commits (including the assigned maintenance categories) per author. Its purpose is to analyze the profiles of developers in the project. For example if an author is responsible for perfective maintenance or if perfective maintenance is distributed evenly on the team.

Dictionary-Report This report provides required information to create and improve dictionaries by showing statistical information for every unique word found in any of the parsed commit messages. The report provides the lemma for the word, the average number of appearances of the word in the commit messages it was found in, the total number of appearances in all the commit messages, the total number of classified and unclassified commits the word was found in

Lemma-Report The lemma-report is the second required report for creating and improving dictionaries. It includes an entry for every unique lemma, together with the number of classified and unclassified commits the lemma was found in

Modules-Report This report shows categorization statistics about the modules of a project. Module structure to be analyzed can be parametrized. E.g. the project has the structure of /util/login/security. We configure a module depth of 2. There will be a row for /util/* and one row for util/login/* in the report

Control-Report This report was used to manually validate the analysis result during our research. It contains every original commit message and the category it was assigned to

6.3 Generation of a Cross-Project Valid Dictionary

To build a representative dictionary, a set of projects to provide initial keywords and to train our dictionary are required. We further need another set of projects to ensure cross-project validity of the dictionary.

Criteria and Selection of Open Source Projects

Eight open source projects were chosen to build, test and cross verify the dictionary. The following criteria were used to select the projects:

Number of commits For our analysis we only considered projects with at least 30,000 commits of code to the code base

App. Name	App. Type	# Devs	# Commits
Enlightenment	Window Manager	187	51,884
Evolution	E-Mail-Client	431	37,500
Firebird	RDBMS	43	51,509
GCC	Compiler-Suite	426	102,672
Python	Interpreter	216	83,100
Wireshark	Packet Analyzer	43	34,067
FreeBSD	OS	536	150,595
Boost	Prog. Library	294	63,616

Table 6.1: Key Figures of the Analyzed Open Source Projects

Number of Developers To show the categorization of the developer role in a project and also to increase the variance of different commit message style only projects with at least 30 developers (distinct author names in the repository log) are considered.

Subversion Repository Our approach is currently based on Subversion repositories. Therefore only projects with access to their Subversion repositories are included.

Table 6.1 shows the key figures of the selected projects.

Populating the dictionary

As a starting point to create the dictionary we analyzed the log of the FreeBSD-Project and used exemplary keywords from prior work (see [19] and [18]) for the categorization. In the next step we ranked the keywords by occurrence. The top three ranked keywords of each category are included in the first dictionary:

Corrective: fix, bug, problem

Adaptive: new, change, patch

Perfective: style, move, removal

This first dictionary constituted the "seed" to create a more exhaustive dictionary. This initial dictionary only categorized a low number of commits, leaving a large number of commits uncategorized. Starting with this seed, we set up an algorithm with the goal to increase the ratio of classified commits to 80% while maintaining adequate values for a self-evaluated precision (0.8) and recall (0.8). Values beyond these thresholds yield diminishing results - either less commits will be classified, or precision and recall will suffer. An early attempt at the algorithm had to be abandoned, because of a too conservative approach in adding words to the dictionary

(stagnation at about 65% of categorized commits). For the final algorithm we used a more open and flexible approach so that more words would qualify for the dictionary. We further introduced weighting of keywords and rulesets for ambiguous, yet strongly indicative words. The following list describes step wise our final algorithm to create the dictionary:

1. Classify the commit using the ”seed” dictionary
2. If the total percentage of classified commits is greater than 80%, EXIT
3. Count the appearances of all words in the commit messages of the non-classified commits and order them by frequency
4. Choose a set of words from the top of the list and add these as a test set to the existing dictionary
5. Count the number of appearances of every word in the test set in each category
6. If the number of appearances of a word in a category is at least 1.5 times of the appearances of the same word in the other categories, add it to the dictionary with a weight of 2 and remove it from the test set
7. If the number of appearances of a word in two classes is at least 1.5 times of the appearances of the same word in the third class, add it to the dictionary to both classes with a weight of 1 and remove it from the test set
8. If neither 6 or 7 are true, remove the word from the test set and do not add it to the dictionary
9. Go to Step 2

This algorithm achieved a classification rate of 80.34 % after 21 iterations on the FreeBSD project. The output is the following dictionary (weights of keywords in brackets, default weight 1).

Corrective: active, against, already, bad, block, bug, build, call, case, catch, cause(2), character, compile, correctly, create, different, dump, error(2), except, exist, explicitly, fail, failure(2), fast, fix(2), format, good, hack, hard, help, init, instead, introduce, issue, lock, log, logic, look, merge, miss(2), null(2), oops(2), operation, operations, pass, previous, previously, probably, problem, properly, random, recent, request, reset, review, run, safe, set, similar, simplify, special, test, think, try, turn, valid, wait, warn(2), warning, wrong(2)

Class	MR	%	Recall	Precision
Corrective	54,015	35.86%	0.92	0.85
Adaptive	56,046	37.21%	0.91	0.80
Perfective	8,484	5.63%	0.86	0.80

Table 6.2: Recall and precision of the classification for the FreeBSD-project

Adaptive: active, add(2), additional(2), against, already, appropriate(2), available(2), bad, behavior, block, build, call, case, catch, change(2), character, compatibility(2), compile, config(2), configuration(2), context(2), correctly, create, currently(2), default(2), different, documentation(2), dump, easier(2), except, exist, explicitly, fail, fast, feature(2), format, future(2), good, hack, hard, header, help, include, information(2), init, inline, install(2), instead, internal(2), introduce, issue, lock, log, logic, look, merge, method(2), necessary(2), new (2), old(2), operation, operations, pass, patch(2), previous, previously, probably, properly, protocol(2) provide(2), random, recent, release(2), replace(2) ,request, require(2), reset, review, run, safe, security(2), set, similar, simple(2), simplify, special, structure(2), switch(2), test, text(2), think, trunk(2), try, turn, useful(2), user(2), valid, version(2), wait

Perfective: cleanup(2), consistent(2), declaration(2), definition(2), header, include, inline, move(2), prototype(2), removal(2), static(2), style(2), unused(2), variable(2), warning, whitespace(2)

Blacklist: cvs2svn, cvs, svn

The analysis further showed that the word "documentation" was assigned to the adaptive category by the algorithm. Since "documentation" is a perfective task per definition, the word "documentation" was moved manually from **adaptive** back to **perfective**. The implications warrant further research however.

This final dictionary was used to classify the FreeBSD-project again and precision and recall were measured based on modification records (MR) as shown in Table 6.2.

We then used the dictionary and the algorithm on the "Boost" project (inital classification rate 74.94%), thereafter on "Enlightenment" project (initial classfication rate 72.80%) and altered the dictionary until it achieved 80% of classified changes. We decided to train the dictionary on two other projects to achieve a greater classification ratio and to work out project-individual language issues (e.g. ambiguously connotated lemmas). After this training phase, the dictionary was used with the "Evolution", "Firebird", "GCC", "Python" and "Wireshark" projects and scored a classification rate of over 80% for each project, without adaption.

Project	# MR	Recall	Precision
Enlightenment	51,884	0.90	0.80
Evolution	37,500	0.96	0.92
Firebird	51,509	0.95	0.90
GCC	102,672	0.92	0.83
Python	83,100	0.93	0.85
Wireshark	34,067	0.92	0.85
FreeBSD	150,595	0.90	0.82
Boost	63,616	0.94	0.88

Table 6.3: Recall and precision of the analysis for various open source projects

6.4 Evaluation of the Dictionary

To evaluate our results, we did a survey with five professional Software Developers. The developers are working for different companies since between two to five years (2,2,4,4 and 5). Our survey was structured as follows:

- **Round 1:** Five questionnaires, each with the 21 changes in the code (7 of each category).
- **Round 2:** Five questionnaires, each with the code changes from Round 1 and with the corresponding commit messages for each change

Inter-rater Agreement

To measure inter-rater Agreement of the developers, we used Fleiss' Kappa on six commits that were identical in each questionnaire. Table 6.4 shows the agreement amongst the developers for these six commits (two commits per category). The resulting Fleiss' Kappa for this matrix is $\boldsymbol{K = 0.48}$. This indicates a **moderate agreement** according to Landis and Koch's Benchmark [110] between the developers themselves.

Commit/Category	**Adap.**	**Corr.**	**Perf.**
Corr. 1	1.0	4.0	0.0
Corr. 2	0.5	4.5	0.0
Perf. 1	1.0	2.0	2.0
Perf. 2	0.0	0.0	5.0
Adap. 1	4.5	0.0	0.5
Adap. 2	4.0	0.0	1.0

Table 6.4: Matrix showing the agreements amongst the developers for the six common commits in evaluation round two

If a commit is assigned to two categories, its count is split between the categories.

Conducting the Evaluation

We conducted the survey in two rounds. Table 6.5 and Table 6.6 show the agreement between developers and the automated classification tool. If a developer chose two categories, a point was split between these categories.

	Automated Classification			
Developers	**Adap.**	**Corr.**	**Perf.**	
Adaptive	**11.0**	4.5	0.5	16.0
Corrective	4.5	**12.0**	0.5	17.0
Perfective	7.5	8.5	**24.0**	40.0
	23.0	25.0	25.0	**47.0**

Table 6.5: Agreements Between Developers and Classification Tool - Evaluation Round One

	Automated Classification			
Developers	**Adap.**	**Corr.**	**Perf.**	
Adaptive	**12.0**	1.5	0.0	13.5
Corrective	3.5	**19.0**	1.0	23.5
Perfective	8.5	4.5	**24.0**	37.0
	24.0	25.0	25.0	**55.0**

Table 6.6: Agreements Between Developers and Classification Tool - Evaluation Round Two

Table 6.7 shows the summarized results of the evaluation rounds one and two. The columns show the total number of agreements between the developers and the automated classifications for each category and the Cohen's Kappa-value.

	Agreement			
Round	**Adap.**	**Corr.**	**Perf.**	**Kappa**
Round 1	11	12	24	0.46
Round 2	12	19	24	0.61

Table 6.7: Comparison of Evaluation Rounds One and Two

Interpretation of the Evaluation

The following conclusions can be drawn from these results:

- Both the agreements in the adaptive category as well as the agreements in the perfective category stayed constant for both rounds. In contrast, the number of agreements for the corrective category has significantly risen between round one and two. From this fact we conclude that corrective maintenance tasks are most difficult to spot just by looking at the source code and without reading the commit message
- The number of agreements for the perfective category is almost perfect in both rounds. We therefore conclude that our classification tool excels at identifying perfective maintenance tasks (a finding similar to Mockus et al's inspection change finding in [37])
- The Kappa-value has risen from 0.46 to 0.61 from round one to round two. This means that with the additional information of the commit message, the developers have converged their decisions with the decisions of the automated classification. Curiously this affected mainly corrective changes
- 0.46 and 0.61 both indicate a **moderate agreement** according to the El Emam Benchmark - see "SPICE Software Process Assessment Kappa benchmark" as introduced in[111]

6.5 Conclusion

The presented study implements and uses one part of the SubCat methodology for cross-project analysis of software evolution based on an automated classification. To achieve these two goals, we completed the following tasks:

Classification Algorithm We developed a classification algorithm which uses a dictionary as its base of decision-making. The classification algorithm uses a set of commits as its input and returns an assignment between the commits and the categories that are defined in the dictionary. It is based on the analysis of the natural language in the commit messages and follows a lexical approach.

Dictionary We presented a dictionary for our classification algorithm that is capable of assigning commits to Swanson's maintenance categories. The cross project validity of the dictionary has been proven on five different open source software projects. We instantly reached a percentage of successfully classified commits of over 80% for each of the projects, without having to adopt the dictionary.

Evaluation We evaluated the dictionary and the automated classification by using a two-step evaluation process. In a first step we evaluated the decisions of the automated classification and the dictionary against our own manual classification. We reached an average

recall of **0.93** and an average precision of **0.86**. In the second evaluation step we evaluated the dictionary against the opinion of five professional software developers. We have proven a **moderate agreement** between the decisions of the automated classification and the decisions of the developers. This result is similar to the result achieved by Mockus et al. in [37] and proves that the approach presented is valid for cross-project analysis in the open source project landscape.

The successful evaluation of the lexical-approach on generic projects has many implications. Researchers can use the implementation of SubCat for a new definition of maintenance and software evolution metrics, not only in open-source projects, but also in any project using non-obscure commit messages. Or fellow researchers can use the implementation to comfortably analyze developer profiles over time in open source projects, e.g. which developer does the bug fixing, who is implementing the new features. Additionally to the main purpose, the categorization of changes into software maintenance categories, the implementation may be easily adapted (by changing the dictionary) for any other studies on commit messages in repositories. The author and dictionary report and to some extent the lemma report are especially useful for this purpose.

CHAPTER 7

Applying the SubCat Methodology for Security Analysis

Contents

The following section was partly published by Mauczka et al. in [38] in the challenge track of MSR to showcase one possible application of the SubCat methodology in security analysis.

7.1 Introduction

Repositories allow for valuable insights on the evolution of software projects. Current research focuses on maintenance related tasks (i.e. bug fixing, bug induction, refactoring - see e.g. [112], [113], [20] or [114]). For the following study, the customizable NLP functionality of an implementation of the SubCat methodology was used to focus on security aspects of software evolution. Examples of related studies have been performed by Gegick et al. [115] and Neuhaus et al. [21]. By performing an analysis on historical change data of the repository, we want to identify security critical changes and find out more about security aspects of a software project. We analyze the SVN repository using a lexical approach that only finds security relevant changes. We plot the security changes on each selected module.

We want to address the following mining questions in the area of change analysis on the FreeBSD project data by using parts of the SubCat methodology:

- Is it possible to identify security related commits (we refer to these as security changes) in the revision control repository
- Is it possible to identify trends in security evolution or are security changes constant?
- Is it possible to differentiate between a security critical module and a regular module based on change history?
- Is it possible to link between security advisories and the revision control repository?

7.2 Problem

Security tends to be a neglected aspect when developing software (see Oram et al. in [116]). Attempts to analyze the security evolution of a project over time seem to focus on vulnerabilities using version history to identify relevant source files and perform further analysis only on source files (see Neuhaus et al. in [21]). There is a lack of exploratory research on the evolution of security and the part of security-related tasks in software development. By using our technique we hope to shed some light on the amount of security effort, measured in security related changes to the software. We try to put security advisories[1] into perspective with the data that can be gathered by the repository to show how accurate the advisories reflect reality and if the security advisories are somehow linked to the repository. Further, it is often hard to pin-point security relevant modules of a software. By using a historic view, we want to find out whether we can determine security critical modules without any specific project knowledge by applying the SubCat methodology.

Questions Addressed

By implementing the SubCat methodology to filter security relevant changes, we want to provide an insight into the evolution of security in modules and to find characteristics of security critical modules - without an intimate knowledge of the code, the processes or the architecture of the system. Furthermore it is important to find out how security is generally treated in a software project. For example if it is added like a feature, late in the software life cycle, or if it is something that is built into the design and thus shows a constant rate of change. We want to find out just how the number of security changes relates to all changes to a module and if it is

[1] `http://security.freebsd.org/advisories.html`

possible to identify modules that are security critical based on revision control repository data alone.

Input Data

The input data is gathered from the SVN repository of the FreeBSD project and the security advisories. We analyze the SVN log and categorize commits based on a lexical approach into security relevant and security irrelevant changes. The procedure is similar to the work of Mockus et al. in [37]. We gathered the data for each directory and ran our analysis. We define a security change as a commit to the SVN that holds a change in a security relevant (e.g. activities related to information disclosure or a fix of a vulnerability) context - we do not classify the security changes themselves into corrective or perfective tasks. To validate the security changes we classify, we use the input gathered from the FreeBSD security advisory[2], which we used as input for a script to congregate the issues on a monthly base. We use this data to validate our approach and to find out whether FreeBSD references the issues tracked in the advisories from the repository.

Approach and Tools

We undertook two analyses and used two different dictionaries (i.e. keywords) as input to the classifying algorithm. One dictionary included a wide variety of security related keywords while the second dictionary was much more conservative and held only two keywords ("security" and "vulnerability"). After using both dictionaries, the conservative dictionary was chosen for the analysis as the other dictionary generated a large number of false positives (e.g. the word "overflow" is often used in a non-security related context). We used a third dictionary consisting of all references in the advisories to find out whether the SVN and the advisories are linked via commit messages.

We then used scripts to identify and collect the security related changes. We grouped changes by month to try to find out more about the historical evolution of the security aspect of software projects. For the follow-up analysis we analyzed the size of the module relating to the number of changes. We gathered relative and absolute measures. We were able to identify modules that implement security critical features.

To gather the data, the SubCat methodology was implemented using a specific domain centered dictionary, in this case a security-term based dictionary. The presented approach shows the

[2] `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/security-advisories.html`

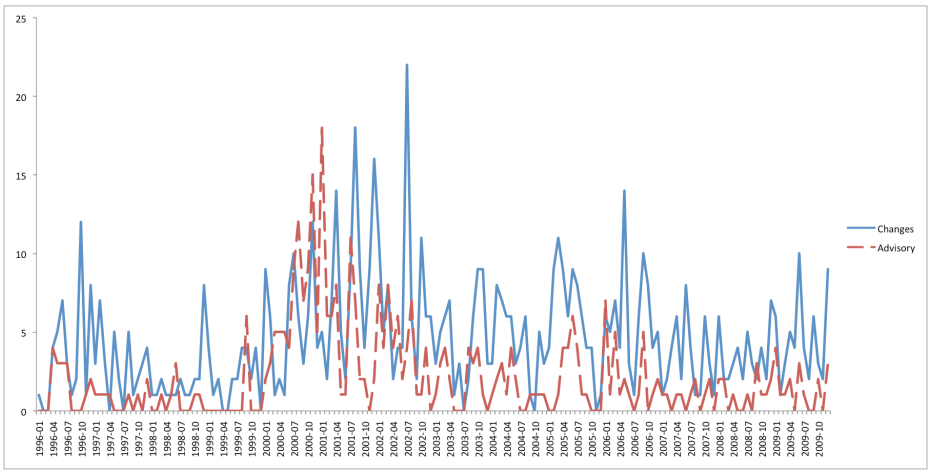


Figure 7.1: Security Evolution - Number of Security Changes and Security Advisories

flexibility and ease of implementation of the SubCat methodology to analyze various aspects of software evolution.

Validation of Results

We validate our data by filling our dictionary with the names of the FreeBSD advisories and finding all security changes containing the advisory ID. Figure 7.2 shows the relationship of found commit messages and reported advisories. A similar approach has been used by Neuhaus et al. [21] to find security bugs. Figure 7.2 shows a policy change of some sort for security-related SVN commits beginning with 2005 and the security changes are now referenced almost 1:1 in the SVN.

To find out whether our scripts classify changes properly we compared changes found by advisory ID with changes found by our keyword list (see figure 7.1). Some commits address more than one security advisory, which accounts for the tendency of the SVN curve to be below the advisory curve. The scripts therefore deliver correct results and the dictionary is correctly measured up against the commit message.

7.3 Results

The first result of our analysis is the number of commits to the FreeBSD repository (Time span starting from directory creation until December 2009). For better visibility we removed directories with zero security changes and we removed the `sys` - directory from figure 7.3, because it skews visibility. It holds the maximum number of overall commits and overall security changes,

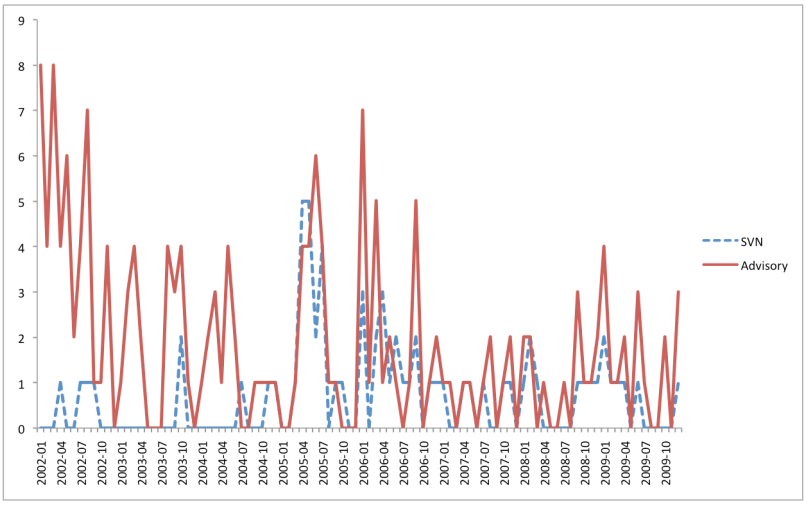


Figure 7.2: Validation - Advisories Referenced by SVN Commit Messages and Reported Advisories

but is also an outlier (see table 7.1). As can be seen in figure 7.3, there are three distinct areas visible - directories with a low number of commits and a low number of security changes, medium to high number of commits and average number of security changes and medium to high number of commits and high number of security changes.

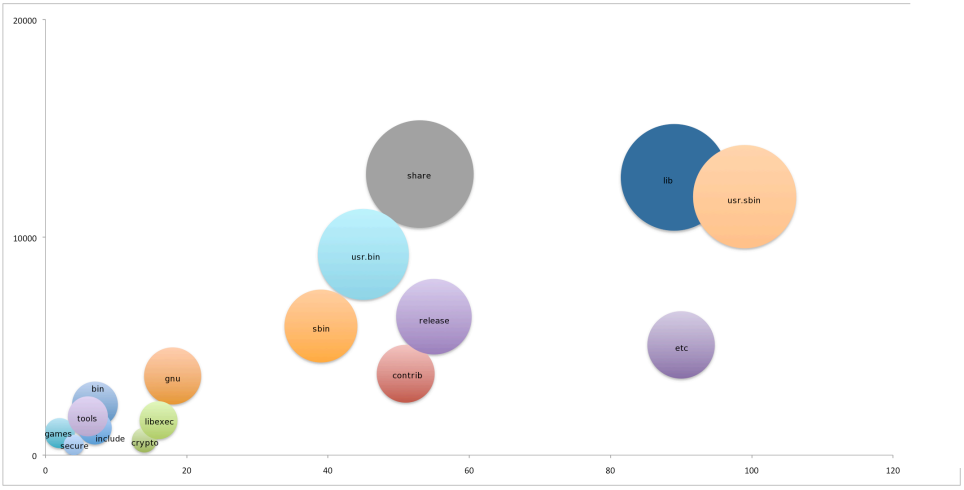


Figure 7.3: Size of Modules - x-axis: Security Commits; y-axis: Commits in Absolute Numbers

The real percentage of security related changes to the overall number of changes is difficult to read from the diagram. However, the diagram visualizes that the assumption that security changes scale somehow with the number of overall commits is viable. Figure 7.4 shows the

Table 7.1: FreeBSD Directories and Change Data Gathered by the Presented Approach

Module	Changes	Security Changes	%
games	1009	2	0,19
bin	2328	7	0,3
tools	1780	6	0,34
share	12905	53	0,41
sys	74953	319	0,43
usr.bin	9217	45	0,49
gnu	3635	18	0,5
include	1255	7	0,56
sbin	5927	39	0,66
lib	12744	89	0,7
usr.sbin	11868	99	0,83
secure	465	4	0,86
release	6356	55	0,87
libexec	1603	16	1,0
contrib	3737	51	1,36
etc	5061	90	1,78
crypto	694	14	2,02

percentage of security changes of all changes from the SVN repository.

Interpretation

To argument the interpretation we used the documentation of the source directory structure[3] and the documentation of the FreeBSD file system structure[4] and figured out the security relevant directories and compared this with our analysis. The directory descriptions cited are from these two documentation pages.

The analysis showed different aspects about security changes within the FreeBSD project. Figure 7.3 shows the number of security changes and absolute number of changes per package. We expected a relationship between the total number of changes and security changes. The package etc is a medium sized package but is related to other packages with more security relevant changes. This leads to the assumption that many security relevant fixes are problems in the configuration. This assumption is further reinforced by both the Open Web Application Security Project (OWASP) with the Top Ten[5] most web application vulnerabilities and the CWE/SANS TOP 25 Most Dangerous Programming Errors[6] contain problems related to etc in their list-

[3] http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/introduction-layout.html

[4] http://www.freebsd.org/doc/handbook/dirstructure.html

[5] http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[6] http://www.sans.org/top25-programming-errors/

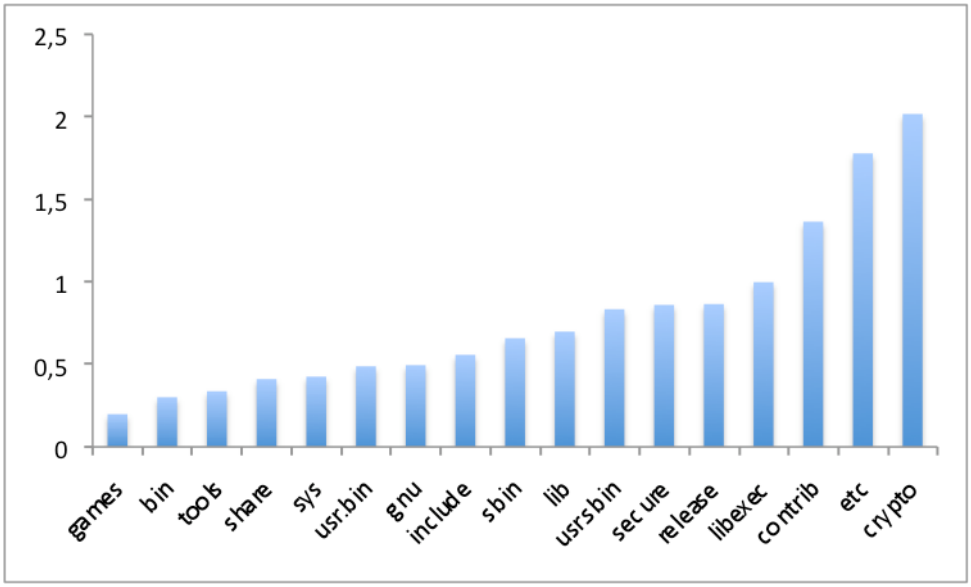


Figure 7.4: Security Proportion - Percentage of Security Changes per Directory

ings. For example “CWE-732: Incorrect Permission Assignment for Critical Resource” occurs several times in SVN commit messages. After filtering out false-positives we still have a >1% relationship.

Another package of interest for our analysis is `usr.sbin` which has, compared to `usr.bin`, more security relevant changes. One reason for that might be that `usr.sbin` contains more security relevant applications (“System daemons & system utilities (executed by users).”) whereas `usr.bin` contains applications for general usage in FreeBSD (“Common utilities, programming tools, and applications.”). Similar to `usr.sbin` is the package of `sbin` (“System programs and administration utilities fundamental to both single-user and multi-user environments.”) and `libexec` (“System daemons & system utilities (executed by other programs).”). Similar to `usr.bin` are other non security critical packages like `bin` (“User utilities fundamental to both single-user and multi-user environments”) or `tools` (“Tools used for maintenance and testing of FreeBSD”). This shows the difference in the analysis between security relevant and non security relevant packages.

Similar to `usr.sbin` we interpret the number of security changes in `lib`. `lib` contains general functionality and will be used by normal and security related applications. A further analysis showed that most of the security changes (56%) are contained in `lib/libc` which is used by many applications.

The package `release` is a non security critical package (“Files required to produce a FreeBSD release”). With this package the lexical approach failed. After manual checking this

package many keywords are used commonly within this package. This package for example contains the release notes and therefore a lot of SVN commit messages contain parts of security advisories added to the release notes. About 66% of detected changes are within the `release/doc` or `release/texts` subpackage.

`contrib` is defined as "consist of software that is actively being maintained outside the FreeBSD project"[7]. It contains some critical applications like `contrib/telnet`, `contrib/bind9` (Domain Name Service) and `contrib/ntp` (Network Time Protocol). After further analyzing all 23 subpackages we found that nearly 25% of all security changes are related to these three subpackages.

After the analysis we can see a threshold of 0.5% of relative security commits (see figure 7.4) that differentiates between security relevant and non security relevant components. Packages with less than 10 changes are only partly mentioned, since the bias by a single security change would be too great (worse, a false positive would have an even greater impact).

7.4 Conclusion

The presented approach showcases how SubCat's generic approach to software evolution research may be customized for a very specific aspect and how insights may be used to identify previously unknown security relevant modules. Using it, we can find security related commits for the FreeBSD project and we can differentiate between a security critical module and a regular module based on change history. For the FreeBSD project we can establish a link between published advisories and the VCS after year 2005, because all advisory related commit messages after year 2005 contain the advisory ID. During the analysis we could not find security trends using security change history for the FreeBSD project.

[7] `http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/policies-contributed.html`

CHAPTER 8

Applying the SubCat Methodology to Populate an Issue Tracker

Contents

Parts of the following section have been published as a technical report [41]. It describes a study on how to apply the SubCat methodology to extract and classify repository data and use this information to populate an issue tracker from an existing repository.

8.1 Introduction

The following study showcases how the SubCat methodology and its generic framework may be extended by scripting tools to provide a completely different usage scenario of software evolution research data. Theoretical approaches reflected in the SubCat methodology are exploited for a very practical goal, the re-population of a BTS from the meta-data available in the VCS.

Combining information between repositories is not a new idea and fundamental to SubCat's design and zhere are many studies on tools and techniques to link different types of repositories together. The importance of this has been pointed out by Hassan in [76]. However, even before Hassan argued for the linking of diverse repositories, there existed tools and approaches based on more than one repository. In [9], Bevan et al. present Kenyon, a tool for source configuration management data mining. They present results for CVS and SVN source code repositories. While that study does not represent a case of different repository types, it does cover different types of source code repositories.

In their work on the tool EPOSee [33], Burch et al. present a tool to analyze software evolution that is able to visualize sequence and association rules, which is especially useful when considering multiple different repositories to showcase causal changes, etc. However, they only provide a visualization tool and no data mining tool.

In [117], Schröter et al. mine Eclipse bug and version databases to map failures to components. This study is of essential interest for the implementation of the SubCat methodology presented in this section. They use their data to relate code, process and developers to defects. From this basic data, different analyses might be taken. In their paper, they use this data to predict failure-proneness or the relationship of bug find rates during testing and release. They employ a simple approach to finding bugs, by looking for bug tracker IDs and certain keywords. We will make an argument for a more sophisticated approach later in this chapter, when we discuss how this approach might not be sufficient to produce desirable results.

Anbalagan and Vouk in [118] discuss a tool based on web-scraping that retrieves data from Bugzilla and the Launchpad bug tracker and compare this data with the National Vulnerability Database. They show that by identifying data in one information they could support mining activity in other repositories. Neuhaus et al. in [119] use information from vulnerability databases and VCS to identify past vulnerabilities in components. They implemented the Vulture tool to automate this process and included a predictor that correctly predicted about half of all vulnerable components.

While there are many studies for tools and techniques to link repository information, automated generation of artifacts from one repository type for another is a novel and practical approach.

8.2 Problem

Issue trackers and source repositories are often connected to each other. This goes as far as commit hooks with task IDs so traceability is possible from requirement/bug report to resolved issue to code base. However, not every project starts out with a perfectly set up application

life-cycle management (ALM). Recurring issue information is lost, as is the ability to browse historical, possibly connected issues. If only the VCS is available, one is naturally restricted to a more modularized view when bugs are analyzed. We use the workflow described in [90] by Sliwerski et al. to establish bug histories for bug related changes. In the following study [120] by Kim et al. an improved version of the SZZ algorithm is presented to automatically identify bug introducing changes. For this study, we assume that no issue tracker is existing yet though, so we use the dictionary generated in chapter 6 to identify bug-fixing commits. We formulate the following research question as main question for the study:

- Is it possible to leverage an improved SZZ algorithm and SubCat to generate meaningful issue reports, populate an issue tracker with the information and thus improve future maintenance tasks by providing meta-information available in Bugzilla?

A different aspect that is rarely analyzed is the delta between bugs occurring in the code base and bugs reported in the issue tracker. In [90], Sliwerski et al. use bug tracker IDs and URL structures to establish a link between bugs in the code base and the bug tracker. They propose that they increase the precision of finding proper bugs in comparison to Mockus and Votta in [18]. On the contrary though, we propose that a lot of bug fixing/introducing is happening during development and test and is never entered into the issue tracker. We will therefore use SubCat to answer the following questions concerning bugs in the code base and bugs reported in an issue tracker:

- What is the ratio between bug-fixing changes as identified by a bug tracker and hence the SZZ algorithm and as identified by SubCat?
- What is the improvement by using the generated dictionary from 6 over naive dictionaries in previous research against known bugs, e.g. bugs that have been reported in the issue tracker?

Furthermore, the developed tooling landscape may be used to populate a Bugzilla BTS for any project using SVN as repository and sensible commit messages.

8.3 Approach and Tools

The approach presented in this study is based on the capability to identify commits that fix a previously introduced bug. As discussed earlier, there are two methods based on commit messages to identify these kind of commits. One is a lexical categorization method and based on the SubCat methodology, presented initially by Mockus in [18], reworked by Hassan in [19]

for open source projects and refined for cross-project validity in [11] and previously in chapter 6. The second method is based on extracting BTS information from commit messages, e.g. an issue ID, or a BTS URL (see [90], [25] and [121]). Since the primary goal of the study is to populate an BTS from an existing repository, the SubCat methodology based approach is of greater importance for this study. However, the second method provides us with the means to validate the identified potential issues by the lexical procedure. It also allows us to address follow up questions when comparing the primary and the secondary approach.

The planned approach for the study is structured as follows:

1. Preliminary analysis
    a) Evaluate robust and simple dictionaries as well as the dictionary from chapter 6 for effectiveness to identify commits for eligibility in the generation of BTS issues
    b) Correct selection of parameters for the application, e.g. stop words in commit messages, regular expressions for BTS linking
2. Designing the application
    a) Derive architecture and tooling setup from the lessons learned in the preliminary analysis
3. Extending SubCat to populate the BTS
    a) Populating the BTS
    b) Presenting the populated BTS

The application itself gathers data from SVN as VCS and Bugzilla as BTS, though the application's modular design may be used to extend the functionality for other VCS or BTS.

8.4 Preliminary analysis

To develop a stand-alone tool that is capable of mining data from the described setup (Bugzilla as BTS and SVN as VCS) and populate an BTS, it is necessary to evaluate which commits in the VCS are actually eligible as input for issues in the BTS in a preliminary analysis. For the evaluation we use projects in which BTS and VCS are connected by references in the commit messages to each other. Thus we can measure the effectiveness of our approach by comparing the commits we identified (classified as corrective by the classification algorithm) as potential issues in a BTS and the actual issues in the BTS.

Before we can apply mining tools on the data, we need to pre-process the raw repository data. The raw data used for this study is generated from SVN and Bugzilla by the following procedures:

Retrieve raw data from SVN: We use the XML-formatted SVN log by using the SVN command `svn log -v -xml path_to_repository > logfile.xml` We apply this for all revisions (i.e. from revision 1 to HEAD revision)

Retrieve raw data from Bugzilla: We filter for BTS issues in the states *resolved*, *verified* or *closed* with the resolution fixed and use the XML-formatted bug report format provided by Bugzilla's webinterface

To ease later processing and validation, we generate XSD-files for both raw data XML-files.

Bugzilla handles bugs in a defined workflow[1]. For our preliminary analysis, however, only RESOLVED, VERIFIED and CLOSED with the resolution fixed are relevant, since these states are likely to have commits associated.

To perform our preliminary analysis and evaluate the best suited dictionary for the implementation of SubCat, several tools are required. These are:

BugZillaParser: This parser extracts the affected revision from comments in Bugzilla bug reports and then merges this information with the categorized commits generated by SubCat

LogFileParser: This parser extracts Bugzilla meta-information from commit messages and merges them with the categorized commits generated by SubCat

ResourceCombiner: This tool combines the reports generated by the BugzillaParser and the LogFileParser to produce one single output file

The goal of the preliminary analysis is to identify which dictionary is best suited for the categorization of commits, since this will be the major input data. To evaluate the dictionaries and find the best suited that will be used for the implementation of SubCat in the final step of the study (i.e. to populate a BTS using information from a VCS), we need the means to determine a ground truth, i.e. find out which revisions are actually bug fixing revisions. We define a revision as bug fixing, if at least one of the two conditions holds true:

1. There is a BTS reference in the bug-fixing commit message
2. There is a revision reference in the BTS issue

[1] `http://bugzilla.readthedocs.org/en/latest/using.html` - there is also the state CLOSED, which is not shown in the diagram

Before we start the analysis, it is required to select projects to provide training data. Since the preliminary setup is one of validation, not all projects are suitable. We define the following requirements for eligible projects:

1. A project for the preliminary analysis has to use SVN
2. A project for the preliminary analysis has to use Bugzilla
3. A project for the preliminary analysis has to have bug fixing commits linked to Bugzilla issues, so commits may be validated as bug-fixing commits
4. A project for the preliminary analysis has to refer to bug fixing revisions in its Bugzilla issues

Based on these criteria, we select the following projects for the preliminary analysis:

Project	**Commits**	**Bug Tickets**[2]	**Time Period**
Wireshark	38.233	3.824	16.09.1998 - 28.07.2011
Mediawiki	94.332	13.955	27.08.2001 - 12.08.2011
Mozilla Websites	93.611	17.547	01.09.2006 - 04.08.2011

Table 8.1: Selected Open Source Projects for Preliminary Analysis

After selecting an initial test data set, we need to define the order in which we will proceed with our analysis. We may break this process down into four steps. The first step is to classify commits into corrective and non-corrective commits by using the implementation of the SubCat methodology. We need to do this step first, as it will provide us with a data set that we use to link issues from Bugzilla with revisions and commit messages with Bugzilla issues. After we gathered links from the SVN to Bugzilla, we mine the comments in our selected issues for revision IDs to establish a connection directed from Bugzilla to SVN. Once both of these steps are done, we combine the data into one data set, which may be further used to compare the performance of the dictionaries we use with SubCat. See figure 8.1 for the process.

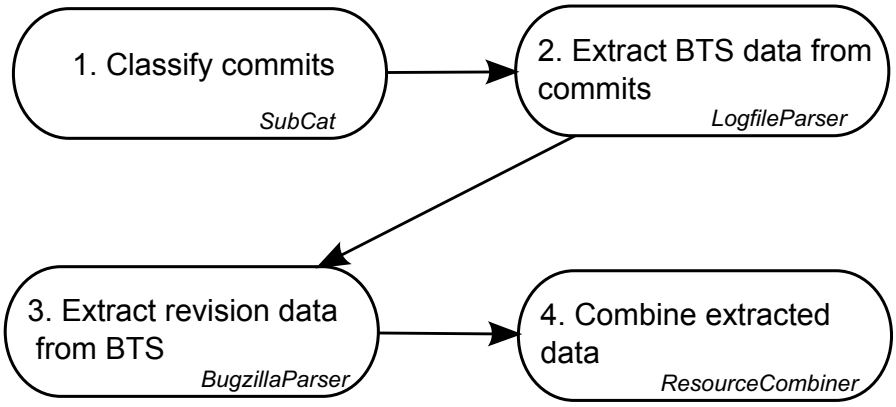


Figure 8.1:
Process for the Preliminary Analysis

The resulting output of the described preliminary analysis process is a report (see table 8.2 for columns and example data), which now may be used to evaluate the efficiency of our dictionaries in finding bugs that are being tracked in a BTS. According to the data of the report, each revision may be classified:

True Positive The revision is correctly categorized as a bug fixing revision - BTS and VCS are uni- or bi-directionally linked

False Negative The revision is not categorized as a bug fixing revision - BTS and VCS are uni- or bi-directionally linked

False Positive The revision is categorized as a bug fixing revision - BTS and VCS are not uni- or bi-directionally linked

True Negative The revision is correctly categorized as a non-bug fixing revision - BTS and VCS are not linked

Rev. ID	Cat.	# add files	# mod. files	# del. files	sum files	BugID VCS	BugID BTS	Severity	Short desc.	Match
1234	corr.	2	5	3	10	101	101	1	Lorem Ipsum	1

Table 8.2: Output of the Preliminary Analysis - Report Header

This data is now generated for each of the dictionaries to be evaluated. Finding the best performing dictionary is crucial in later tool performance, since the implementation needs to detect all bugs a BTS would traditionally track.

Dictionary 1 - Simple Dictionary

The first evaluated dictionary is based on related work by Mockus and Votta in [18] and the previously mentioned study by Sliwerski et al. [90]. The dictionary suggested in these studies contains a small number of terms and does not include any weighting of terms (see table 8.3).

Dictionary 1 (category: bug fixing)
bug, change, fix, problem, patch, correct, support, incorrect

Table 8.3: Simple Dictionary of Terms Associated with Bugs

Dictionary 2 - Robust Dictionary

The second evaluated dictionary is very robust and unambiguous. It contains only two terms and its main objective is to provide a version with very low false positives (see table 8.4).

Dictionary 2 (category: bug fixing)
bug, fix

Table 8.4: Reduced Dictionary of Terms Associated with Bugs

Dictionary 3 - Complex Dictionary

The third dictionary is an interim dictionary from chapter 6 as evaluation of the final dictionary was still going on during this experiment (see table 8.5). It comprises a large number of terms associated with all of Swanson's maintenance categories. However, for this study, only the terms of the dictionary tagged as corrective are of importance. We point out that only the corrective category and the blacklist category of the dictionary were used. This is important, because there might be potential to reduce false positives, since these would be put into different Swanson's categories (enhancement or perfective).

Dictionary 3 (category: bug fixing)
better, right, without, something, message, fixme, sorry, should, dont, work, warning, anymore, possible, workaround, iterator, workarounds, bugfix, line, init, character, random, oops, issue, active, explicitly, fast, simplify, look, build, think, catch, recent, exist, create, failure, review, dump, similar, logic, against, good, except, instead, special, operation, safe, hack, hard, pass, lock, help, bad, operations, request, block, call, introduce, turn, valid, wait, fail, previously, warn, log, probably, reset, cause, case, different, wrong, already, properly, compile, try, format, correctly, miss, test, run, set, error, previous, null,fix, bug, problem

Table 8.5: Complex Dictionary of Terms Associated with Bugs

Results of the Preliminary Analysis

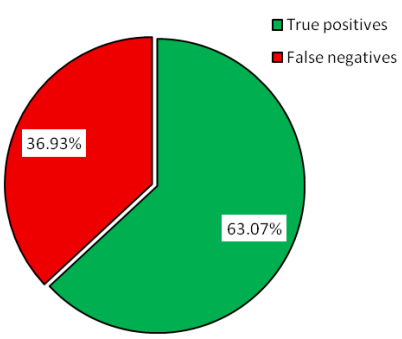
We performed our analysis of the three different dictionaries on various test projects to exclude any obvious project bias on the dictionaries performance. During this analysis we calculated two measures:

1. Correctly detected versus undetected bug fixing revisions
2. Correctly detected versus erroneously detected bug fixing revisions.

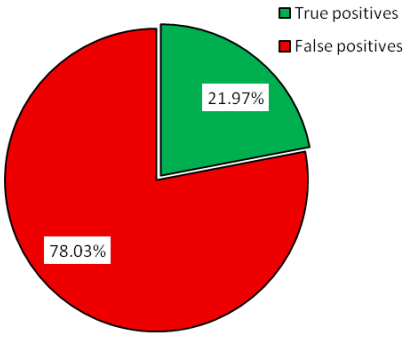
As the first project, Wireshark was selected. At the time of analysis there had been 38.233 revisions. Out of these 38.233 revisions, we could establish a link as bug fixing revisions to the BTS for 4.896. Out of these 4.896 potentially identifiable bug-fixing revisions, 3.088 were correctly identified by the dictionary, while 1.808 were missed. This means that dictionary 1 achieved 63% true positives for detecting referenced bug-fixing revisions and 37% false negatives. In total, dictionary 1 classified 14.054 revisions as bug-fixing, which concludes that 10.966 as bug-fixing classified revisions could not be referenced to the BTS. Dictionary 1 produced 22% true positives for detecting referenced bug-fixing revisions, compared to 78% false positives for all revisions.

Dictionary 2 identified 2.536 out of 4.896 referenced BTS issues correctly as bug-fixing revisions (52% true positives) with 48% false negatives. This is about 10% less than dictionary 1 which was expected. Its still surprising though that with only 2 terms, over 50% of the bug-fixing revisions could be identified. Dictionary 2 marked only 9.237 revisions as bug-fixing revisions. This is about 4.800 revisions less than dictionary 1. This means that 6.701 bug-fixing revisions were erroneously classified compared to the BTS. Interestingly enough, this is an increase in precision compared to dictionary 1.

After the first two runs of dictionary 1 and 2, we investigated our combined data further and found that we identified enhancements in the issue tracker also as bug-fixing revisions. This

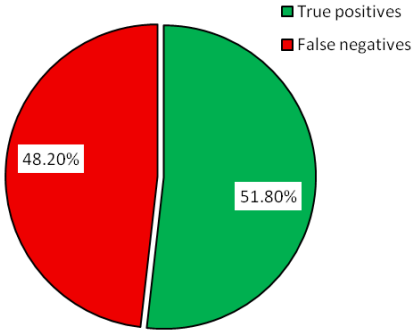


(a)
Measure 1:
Detected vs. undetected

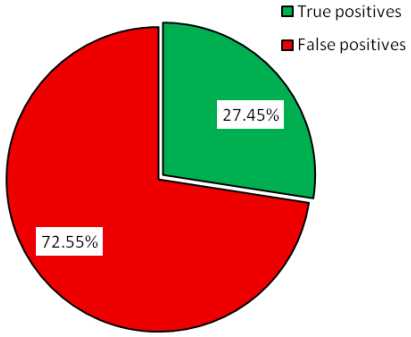


(b)
Measure 2:
Detected vs. erroneously detected

Figure 8.2: Classification of Bug-Fixing Revisions for Dictionary 1 on Wireshark



(a)
Measure 1:
Detected vs. undetected



(b)
Measure 2:
Detected vs. erroneously detected

Figure 8.3: Classification of Bug-Fixing Revisions for Dictionary 2 on Wireshark

showcased a pitfall of using NLP methods on unpruned artifacts - the implementation of the SubCat methodology normalized all commit messages and thus split URLs into single terms. This lead to a misplacing of all referenced enhancement issues into the bug-fixing category. To address this obstacle, we implemented a filter mechanism for URLs, since the previously generated results were obviously erroneous. After removing BTS information from the commit messages, we achieved the results for dictionary 1 and 2 shown in figure 8.4. While the values do shift, the shift is not as great as expected, indicating again that even without referencing by URL, usage of NLP for classifcation is promising. An interesting aspect is that by filtering out the previously misclassified enhancement, both dictionaries improved their recall. Dictionary

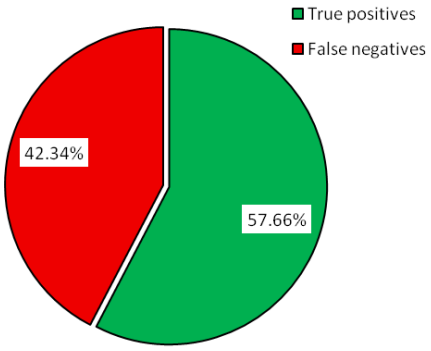
1 performs better when it comes down to detecting relevant BTS entries, which seems better suited for the goal of the final study.

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

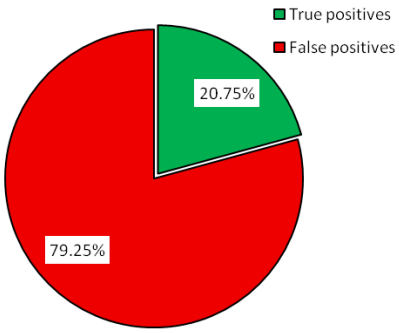
If we compare the F-score calculated for dictionary 1 and dictionary 2, we see that dictionary 2 provides slightly better results. However, as indicated earlier, we prefer a high recall to a high precision.

Dictionary 1: $F_1 = 0.31$

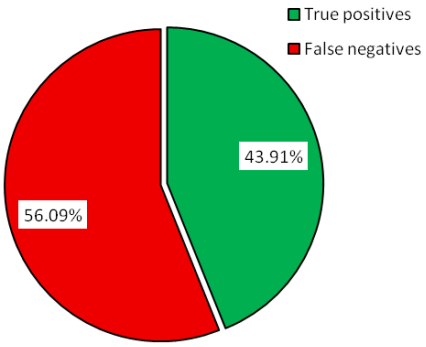
Dictionary 2: $F_1 = 0.32$



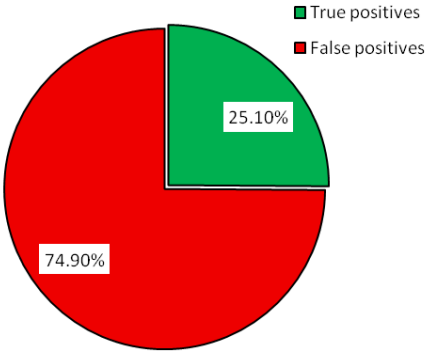
(a)
D1 - Measure 1:
Detected vs. undetected



(b)
D1 - Measure 2:
Detected vs. erroneously detected



(c)
D2 - Measure 1:
Detected vs. undetected

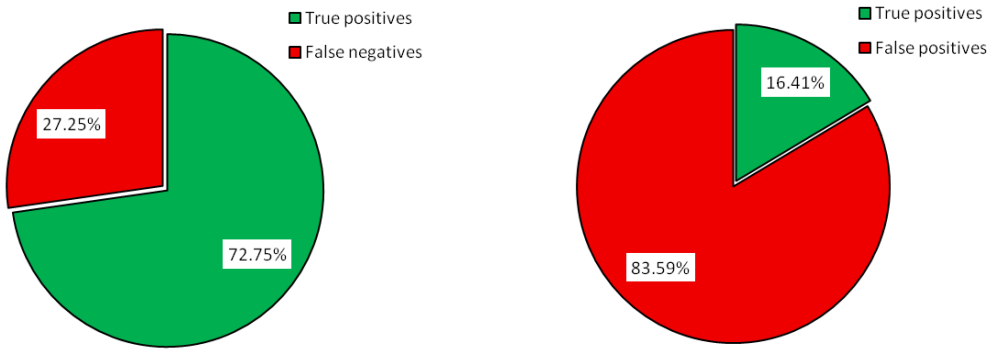


(d)
D2 - Measure 2:
Detected vs. erroneously detected

Figure 8.4: Classification of Bug-Fixing Revisions for Dictionary 1 and 2 on Wireshark Without URLs

After doing the analysis for dictionary 1 and 2, the outputs generated by using dictionary 3 were analyzed for the Wireshark repository data. Since this dictionary was already evaluated in a developer survey to achieve a high classification rate, we expected the dictionary to perform well in recall on the BTS issues. Dictionary 3 detected 3.562 out of 4.896 bug fixing revisions (73%). This is an improvement by 26% to dictionary 1 and almost 66% to dictionary 2. Further, we can safely discard dictionary 2 from future consideration, as there is obviously a great benefit in fine-grained analysis of commit messages. Furthermore, dictionary 3 identified 21.701 bug-fixing revisions, which is an increase by almost 60% to dictionary 1 and 153% to dictionary 2. This also means that 84% of bug-fixing revisions were false positives. This precision resulted in the lowest F-score (0.27) of all 3 dictionaries.

Dictionary 3: $F_1 = 0.27$



(a)
D3 - Measure 1:
Detected vs. undetected

(b)
D3 - Measure 2:
Detected vs. erroneously detected

Figure 8.5: Classification of Bug-Fixing Revisions for Dictionary 3 on Wireshark

After reviewing a sample of the false positive corrective changes (commits that could not be linked to the BTS), we could identify some bug-fixing revisions as false positives, since we could establish a link to enhancement tickets in the BTS. Of the 84% false positives, we could establish 4% as enhancement tickets. Taken the results from chapter 6 into consideration for the final dictionary, where we established a precision for corrective changes of 0.70, recall of 0.92 and identified 10.242 corrective changes, it is safe to assume that of the remaining 80% there are several more false positives. However, since our goal is a high recall for the BTS issues and any other generated bug tickets may be considered useful, we decided to use dictionary 3 for further preliminary investigation. Table 8.6 summarizes the performance of the 3 dictionaries for Wireshark.

Dictionary	# of BTS issues	# of missed BTS issues	# of corr. identified revisions in VCS	# of err. classified revisions in VCS
Dict. 1	3.088	1.808	14.054	10.966
Dict. 2	2.536	2.360	9.237	6.701
Dict. 3	3.562	1.334	21.701	18.139

Table 8.6: Dictionary Performance - Wireshark: 38.233 Revisions; 4.896 Revisions with ref. to BTS Issues

In the next steps of the preliminary analysis, we decided to evaluate dictionary 3 with Mediawiki. The results of the evaluation can be seen in table 8.7 - dictionary 3 identified over 90% of the BTS referenced revisions.

Dictionary	# of ref. BTS issues	# of missed BTS issues	# of corr. identified revisions in VCS	# of all classified revisions in VCS
Mediawiki	7.477	561	6.916	54.055

Table 8.7: Performance of Dictionary 3 on Mediawiki

In conclusion of the preliminary analysis, we discovered that:

1. Many bug-fixing revisions never make it to the BTS
2. The interim complex dictionary classified 90%+ of revisions with reference to the BTS

Finding 1 is especially interesting considering the work of Sliwerski et al. [90] and similar studies focusing on BTS, since it clearly shows there is a lot more bug-fixing going on during software evolution than which can be learned from a BTS. This stresses the importance of combining repositories to research software evolution. A full fledged study on this topic with the final dictionary from chapter 6 might help in understanding the delta between bugs found in the BTS and bugs found by mining commits and the underlying processes for corrective maintenance. Finding 2 is important for the next steps of this experiment, when we use the dictionary to find as many potential tickets as possible to populate the BTS. As indicated earlier, we prefer recall to precision, since BTS offer great filtering and search functionality - the more data we provide, the better.

Before we started with the population of the BTS, we also investigated statistics on number of files affected by a commit, since research on this topic indicated the need for data pruning (see e.g. Hindle et al. [109], [87]). After manually filtering through outliers, we were able to determine four categories for these commits:

1. Copying of revisions between branches
2. Changes to certain SVN properties
3. Signature changes of a method
4. Perfective maintenance

These categories generally did not include corrective maintenance tasks - we then stepwise discarded revisions from the analysis by removing commits with a file count >500, >100, >20,

>10 and >5. After each step, we review the discarded revisions. No corrective changes were discarded until the last step. We repeated this procedure for Mediawiki and found six files per commit as threshold for discarding of revisions in the population of the BTS.

8.5 Designing the Application

The final application consists of two parts, the first part is the command line functionality of SubCat with an interim dictionary to classify corrective changes. The second part encapsulates all the processing logic and includes a slightly altered SZZ algorithm to identify bug introducing changes and an algorithm to trace bug-fixing processes that span more than two revisions, or steps. It also includes the data generator for the BTS. See figure 8.6 for an overview of the architecture.

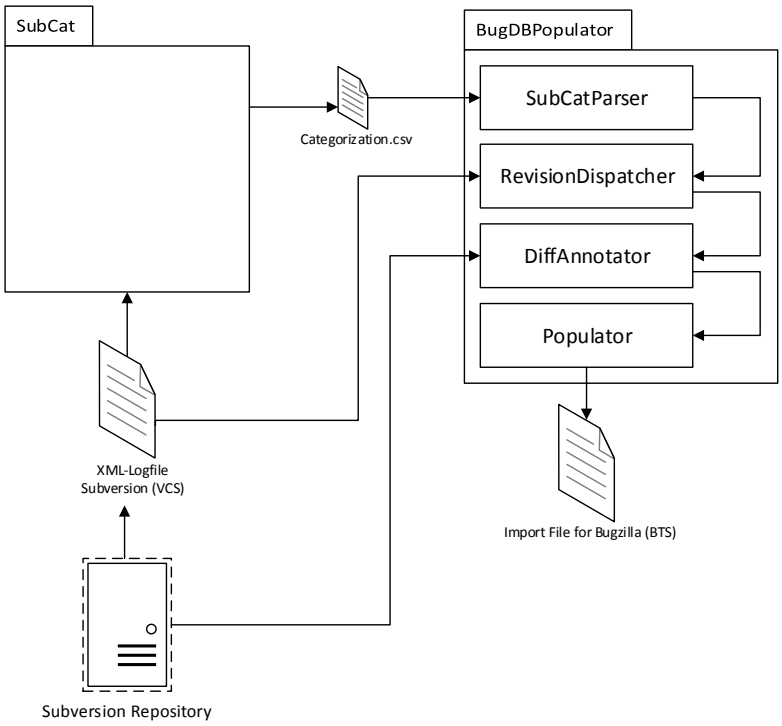


Figure 8.6:
Experimental Setup: Architecture Overview

The BugDBPopulator consists of four major components, which perform most of the tasks that were done manually during our preliminary analysis. In the first step, the *SubCatParser*, we process the categorization output of SubCat and generate the revision data set the other compo-

nents work on. The input can be parametrized so that different file inputs may be possible as long as they are in comma separated value and have columns for revision number and category. The number of files affected in a revision may also be used as a filter. The *RevisionDispatcher* uses the data from the SubCat parser to retrieve file indexes for all revisions (these may also be parametrized to filter unwanted file extensions). These two components are data preparation components. The business logic to mine bug histories and to generate the output for the BTS is handled in the next components. The *DiffAnnotator* performs three tasks for each commit:

1. Retrieve previous (A) and current (B) revision of affected files
2. Generate diff of two consecutive revisions of a file
3. Annotate affected file

If there is more than one revision of a file, we generate a diff of this file. To avoid misidentification of bug-introducing or bug-fixing revisions, we ignore tabs, spaces and empty lines. We also supply the possibility to ignore certain pre-fixed lines (e.g. to avoid comments or framework code). The resulting lines from A are stored as bug-introducing lines and the results from B as bug-fixing lines together with the affected file of the commit. After this is done, *DiffAnnotator* calls `SVN annotate` and passes the affected file and revision B. From the annotation result, *DiffAnnotator* stores each changing revision per annotated line, so a complete map of changed lines per revisions is stored together with the affected file. In some cases a bug-fix will affect several files and several lines, so there is not a single revision that introduced the bug. This means that we assign multiple bug-introducing revisions for a single bug-fixing revision. By implementing this altered form of the SZZ algorithm, we can establish direct links between bug-introducing and bug-fixing changes. However, we are also interested in generating bug-histories before the bug-fixing change. To achieve this, we implement a recursive algorithm that consists of the following six steps (described in [41]):

1. Classify all revisions into bug-fixing and non bug-fixing revisions
2. Determine the corresponding direct bug introducing revision(s) and assign them to the bug-fixing revisions
3. Sort all bug fixing revisions by their revision number
4. For all existing bug fixing revisions
    a) Take all determined bug introducing revisions for the bug-fixing revision under consideration

b) For all bug introducing revisions

   i. Check the classification of the bug introducing revision - if it is a bug fixing revision, assign this revision to the current revision under consideration and go to 4a. - if it is not a bug fixing revision, proceed with step 4c.

c) Proceed to next bug fixing revision and continue with 4a.

This algorithm is related to Purushothaman and Perry in [122] who proposed that "in nearly 60% of cases, lines that are changed were changed again" and to Sliwerskis assumption that there exist "partial fixes". We propose that if the implemented SZZ algorithm links a bug-fixing to a bug introducing change, which in turn is a bug-fixing change, the original bug-fix (and any bug-fix up until the last bug-fix) was only a partial fix, or itself a bug-introducing fix. By exploiting this assumption, we are therefore able to deliver bug-fixing histories beyond a direct relationship of bug-fixing and bug-introducing changes by chaining bug-introducing and bug-fixing changes back to back.

8.6 Extending SubCat to Populate the BTS

After establishing bug histories, the next step in our experimental setup is to determine which of the meta information on issues a BTS provides may be automatically generated by the VCS content. Table 8.8 provides a short field description and a mapping to the VCS data we used to fill this field.

Field Name	Description	Mapping data
Bug ID	Unique ID of an entry in the BTS	Generated unique ID by BugDBPopulator
Short Description	Brief description of the issue, important to give fast overview of issue	Contains the commit message, as well as the path to the file with the most affected lines by the revision of the bug-fixing commit
Status/Resolution	Life-cycle state of the issue	Status is reported as Resolved/Fixed
Assigned to	The person responsible for taking action on the issue	Author of the final bug fixing revision
Reported	Shows when the issue was reported and by whom	Date of first bug-introducing revision
Modified	Last modification time of issue	Date of final bug-fixing revision
Importance	A combination of priority and severity of an issue	Statically set to pre-configured value
Component/Product	Component that harbors the issue if there is more than one	Statically set to pre-configured value
Version	The version of the system that the issue was discovered in	Statically set to pre-configured value
Platform	The platform on which the system was tested	Statically set to pre-configured value

Table 8.8: Mapping of BTS fields to VCS data

To provide the user with further information, we append all revisions in the history of a bug as comments. We use the revision date as creation date for the comment and assign the author of the revision as author of the comment. We note the revision number, the log message for the revision and all affected files including all changed lines. See table 8.9 for an example of a ticket.

Using our automatically generated BTS, we analyzed examples of bug tickets and found that the most common bug tickets were expectedly one-step bug tickets. These are tickets that only include one or more bug-introducing revisions and one bug-fixing revisions. In the simplest case this means one revision with one file affected introduced a bug and one revision fixed the bug. There are more complex cases however, when a fixing revision includes changes to many files and different revisions. Therefore, we were able to derive three kinds of one-step bug tickets during manual inspection:

Atomic bug tickets The simplest case of bug introducing and bug-fixing revision. They commonly include only a few changes to the code and are solely dedicated to fixing the introduced bug

Orphaned bug fixes Some bug tickets only include a bug fixing revision, because there was no change to existing code necessary to fix the problem, so there is no bug-introducing revision to be found

Bug 13 - It's not a good idea to use tcph-⟩th_seglen if you haven't set it.
/trunk/epan/dissectors/packet-tcp.c

Status:	RESOLVED FIXED	**Reported:**	2007-11-08 01:58:38 by gerald
		Modified:	2010-12-31 01:02:54
Product:	Wireshark		
Component:	Wireshark		
Importance:	Low Normal		
Assigned To:	guy		

- - - Reported by gerald 2007-11-08 01:58:38 MESZ - - -

Rev: 23396
Msg: Add more TCP analysis struct checks.

```
/trunk/epan/dissectors/packet-tcp.c
  if(tcpd && (tcph->th_flags & TH_FIN)
```

- - - Additional Comments From guy 2010-12-31 01:02:54 MESZ - - -

Fixed in Rev: 35313
Msg: It's not a good idea to use tcph-⟩th_seglen if you haven't set it.

```
/trunk/epan/dissectors/packet-tcp.c
  if(tcph->th_have_seglen && tcpd && (tcph->th_flags & TH_FIN)
```

Table 8.9: Example 1 of an Automatically Populated Bug Ticket

Overloaded bug tickets A number of bug tickets include bug-fixing revisions that address more than three bug originating revisions. This is usually the case, if more than 10 lines of code are affected. This results in very overloaded bug tickets in the populated BTS. This problem was previously encountered by [123] when they mined annotation graphs.

The most meaningful and human readable bug reports were generated with a maximum of three bug introducing revisions per bug-fixing revision. Table 8.10 shows an example of a ticket with two bug introducing revisions and one bug fixing revision.

Bug 154 - small fixes
/tests/Fest/src/org/gjt/sp/jedit/testframework/EBFixture.java

Status:	RESOLVED FIXED	**Reported:**	2009-06-20 14:33:55 by daleanson
		Modified:	2011-06-26 20:06:36
Product:	jEdit		
Component:	jEdit		
Importance:	Low Normal		
Assigned To:	kerik-sf		

- - - Reported by daleanson 2006-10-22 18:37:05 MESZ - - -
Rev: 15501
Msg: Initial commit of test framework using Fest. See build.xml and example_plugin_test_build.xml for details on using the framework.

```
/tests/Fest/src/org/gjt/sp/jedit/testframework/
 FirstDialogMatcher.java
  if(comp instanceof Dialog){
```

- - - Additional Comments From kerik-sf 2009-10-11 18:18:37 MEZ - - -
Rev: 16318
Msg: - automate all tests meant by test_data
- make GeneralOptionPane and XMLInsert Fest friendly

```
/tests/Fest/src/org/gjt/sp/jedit/testframework/EBFixture.java
  if(condition.matches(message)){
```

- - - Additional Comments From kerik-sf 2011-06-26 20:06:36 MEZ - - -
Fixed in Rev: 19634
Msg: small fixes

```
/tests/Fest/src/org/gjt/sp/jedit/testframework/EBFixture.java
  if(condition == null || condition.matches(message)){

/tests/Fest/src/org/gjt/sp/jedit/testframework/
 FirstDialogMatcher.java
  if(comp instanceof Dialog && comp.isVisible()){
```

Table 8.10: Example 2 of an Automatically Populated Bug Ticket

Additionally to the one-step bug tickets, we also encountered complex bug tickets that resulted from our recursive bug-chaining analysis. On manual inspection, we could also determine

different kinds of these type of tickets:

Dependent changes These type of bug tickets describe partial bug fixes or bug fixes that introduced further bugs into the code. Purushothaman and Perry [122] distinguish here between "error by commission" and "error by ommission". For our experiment, we only encountered errors by commission, e.g. bug-fixing changes that actually introduced new bugs into the code base

Unrelated changes Changes that are of a larger size tend to fix more than one issue. By using our recursive algorithm to find bug chains, we assign bug-introducing changes to bug-fixing changes that are in reality unrelated. A filter for changed lines seems sensible to avoid these type of chained bug histories that do not hold any relevant information

Overloaded bug tickets Similar to the one-step bug tickets, any bug-fix chain that exceeds two steps is overloaded with information and hard to read/understand.

8.7 Conclusion

In the presented study, the SubCat methodology was used to provide a novel approach with classified change data. As can be seen by the examples given in the previous section, the populated bug database allowed us to easily reproduce results of previous studies by Zimmermann [123] and Purushothaman [122]. Furthermore, there are several benefits for software evolution in the presented approach. Populating the BTS with VCS information was very useful when looking for design problems, e.g. a search on buffer overflow or security topics will list all revisions that dealt with these problems. By adding log messages and filenames as comments, we can explicitly search histories of files, even though the SVN history might have been broken. If a project started out without a BTS, our approach definitely delivers a good starting point, since the BTS search capabilities are much more comfortable than the VCS ones. It is easier to see connections between changes also. However, parameter tuning has to be done for every project, especially to restrict the file number by revision and the bug-fixing chains to a level of two. Otherwise the number of overloaded bug tickets might affect any meaningful search functionality of the BTS. Since we partially access the SVN directly during populating the BTS, we suggest to either setup the SVN locally or prepare for a long-running analysis (20.000 revisions last approximately a day).

CHAPTER 9

Applying the SubCat Methodology to Create Developer Profiles

Contents

The following preliminary study showcases an exemplary implementation of the SubCat methodology and how it might be used to generate developer profiles for a dashboard like functionality

9.1 Introduction

One of the key factors for developer motivation in open source projects (and likely in industrial settings as well) is fun, as pointed out by Crowston in [69]. Shah in [70] also defines the hobbyist as an archetype of open source participation. They further state that long term engagement in open source projects can be attributed to the factors fun and enjoyment. Shah also points out that classical maintenance chores are done by hobbyist contributers, which seems repetitious and not enjoyable at all. However, long term enjoyment of a project may also be derived from

the perception of a project in the community, which in turn is largely affected by how well the project is maintained. As pointed out in section 2, Wu et al. in [71] conform some of the hypothesis by Shah, while not mentioning fun or the hobbyist archetype defined by Shah. Fang and Neufeld in [72] also point out that social interaction is important for long term participation in open source projects.

9.2 Problem Description

All of the studies mentioned in the previous section are based on questionnaires and interviews and provide empirical data on developer motivation. Since one of the goals of SubCat is to measure social metrics, i.e. to quantify social activity or to measure the mood expressed in either commit messages or issues/comments on issues of open source project members, we may use SubCat to quantitatively assess the findings of the questionnaires. In the first step, which is also the scope of this preliminary study, we want to assess an open source project by measuring metrics and see if any trends become obvious. Furthermore, since fun is an important factor for long term motivation, we want to see if long term motivation is influenced by the daily chores of a developer, i.e. if his maintenance profile, as measured by SubCat's Swanson's task classifier, can serve as an indication for sentiment expressed in the project. We also try to find out, whether stressful situations in a project, which arguably should reduce the factors fun and enjoyment in the short term of a project, are measurable strictly by social metrics, i.e. is an increase in social interactions and an increase in negative language used in commits and/or issues/comments in the BTS a sign of an impending release (see Guzman et al. for similar research on this topic in [102]).

9.3 Selection of Sample Project

Since we want to measure soft factors like fun and developer task profiles, we select the Vala[1] and the gnome-shell[2] project for our preliminary study. The selection of Vala is mainly attributed to the availability of developers, who will be able to tell us their state of motivation during certain project phases. This is important as it may help in future studies to formulate research questions. Also, we will be able to compare the measured metrics with personal perception of project members and their roles and attitudes described in chapter 10. GNOME shell was chosen as it is one of the most active projects currently on the GNOME Bugzilla[3].

[1] `https://wiki.gnome.org/Projects/Vala`
[2] `https://wiki.gnome.org/Projects/GnomeShell`
[3] `https://bugzilla.gnome.org/page.cgi?id=weekly-bug-summary.html`

The preliminary study uses the current implementation of SubCat capable of mining sentiment and categorized commits. SubCat shall provide metrics on activity in the BTS and VCS as well as task profiles of developers. Furthermore, we provide sentiment and task profiles over time and see if any indications for a correlation may be identified.

9.4 Results

For both the GNOME shell and the Vala project, we gathered the top three BTS and VCS contributors and measured commits in the VCS, interactions in the BTS and word counts of contributions in the BTS in 2014. We used NLP functionality of SubCat to create developer profiles for both projects based on categorized commits. The top committers and top contributors to the BTS were chosen for the survey in both projects and are shown in table 9.1.

Developer	**Project**	**Commits**	**BTS Interactions**
Dev A	Vala	97	2
Dev B	Vala	95	1
Dev C	Vala	75	19
Dev D	Vala	16	27
Dev E	Vala	1	11
Dev F	GNOME shell	182	37
Dev G	GNOME shell	119	11
Dev H	GNOME shell	79	34
Dev I	GNOME shell	8	31

Table 9.1: Selected Developers for Preliminary Study (2014)

We show different types of profiles for the selected contributors generated by SubCat. Figure 9.1 for Vala and figure 9.2 for GNOME shell show profiles for the contributors based on the categorization results and the overall classified commits for 2014.

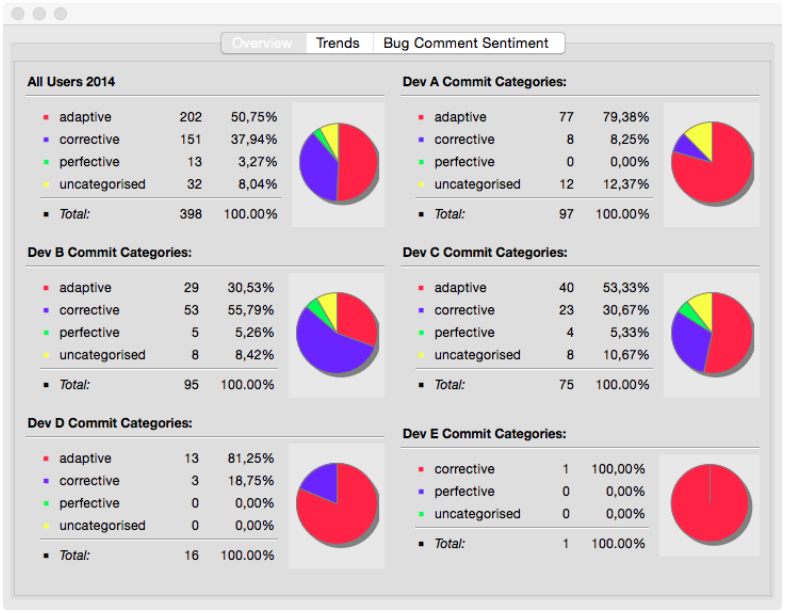


Figure 9.1: Overview Developer Task Profiles Vala

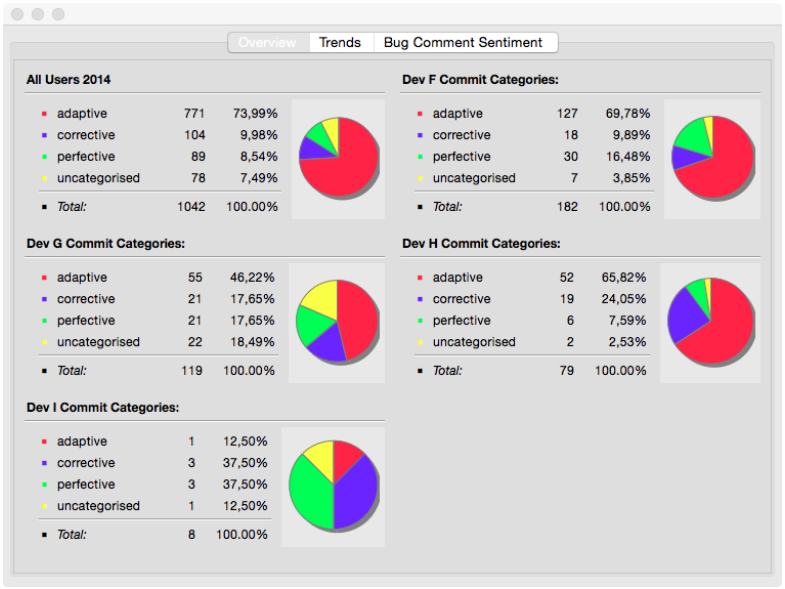
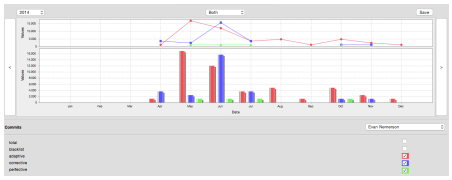
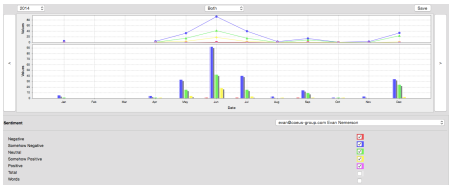


Figure 9.2: Overview Developer Task Profiles GNOME shell

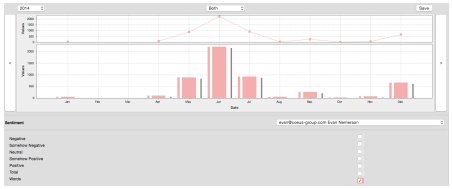
Figure 9.3 shows trend charts for the two top contributers in Vala and GNOME shell based on the largest combined values of commits and BTS activity. These two developers are the most active participants in both repositories and thus their profiles might provide insight on their role in the projects.



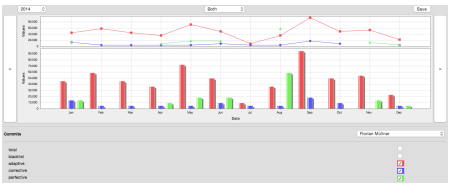
(a)
Commits Over Time



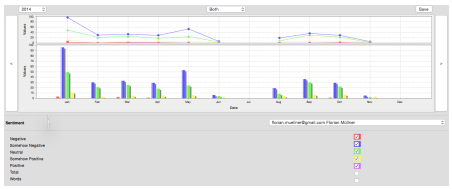
(b)
Sentiment Over Time



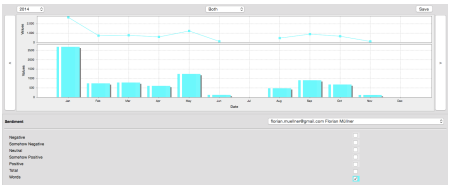
(c)
Activity Over Time



(d)
Commits Over Time



(e)
Sentiment Over Time



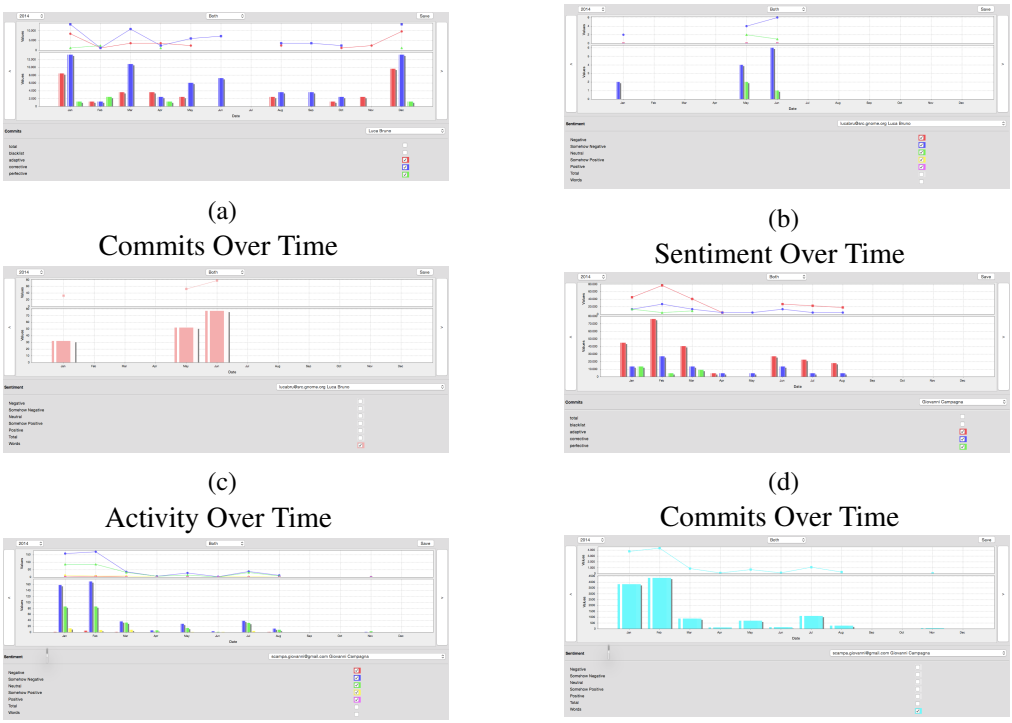
(f)
Activity Over Time

Figure 9.3: Developer C and F: Trends for Tasks, Sentiment and Activity

Other trendcharts of notice are for Developer B, the second most active committer of Vala and Developer H. Developer B shows no activity at all in the BTS, while Developer H seems to have left the project after engaging and contributing to the project in the first months of 2014. See figure 9.4 for details.

9.5 Conclusion

The previous sections showed some interesting profiles for developers in the analyzed projects that warrant further empirical research. Developer B is a very active committer for the Vala project and is hardly active in the BTS. From interviews with the developer, we know that most of his work is triggered by bug reports from Bugzilla, which is reflected in his profile, as he is



(a)
Commits Over Time

(b)
Sentiment Over Time

(c)
Activity Over Time

(d)
Commits Over Time

(e)
Sentiment Over Time

(f)
Activity Over Time

Figure 9.4: Developer B and H: Trends for Tasks, Sentiment and Activity

largely performing corrective tasks. Further, a large part of communication in Vala occurs in the IRC and never enters a repository. Still, his role seems a specialized one which is also not found anywhere else in the analyzed projects. Developer H has largely reduced his contributions to the project by the end of 2014. He was very active and verbal in both VCS and BTS. Expressed sentiment of developer H seems in line with all other developers. This indication warrants further qualitative research as to the reason of the decline in activity. Similarity of profiles based on task categorization to some extent may be found in Developer A and F, as both focus on adaptive tasks. Also profiles for Developer C and H are vaguely similar. This is interesting as they are both the second most active contributer to the BTS (with Developer H being much more active in contributions to the BTS as well as number of words in these contributions). From interviews with Developer F we know that he describes his role as being a maintainer and he is responsible for maintaining language bindings for Vala. His tasks are also triggered by BTS issues, but additionally by updates in dependent APIs, which could make up for the difference in his task profiles to Developer B.

9.6 Future Works

SubCat's implementation provides possibilities to configure and generate views similar to these presented in this preliminary study. We configured SubCat to provide dashboard like information on categorized tasks in the VCS and sentiment as well as activity in the BTS. Even in this small sample, it was possible to identify distinct different profiles (e.g. Developer A, Developer C and F). The next step is to gather similar data with SubCat and use it in an empirical study that assesses these profiles and identifies corresponding roles using qualitative and quantitative methods.

CHAPTER 10

Applying the SubCat Methodology in a Survey for Commit Classification

Contents

The following section has been published in [39] as part of the data set track of MSR to showcase one possible application of the mining framework, which is part of the SubCat methodology, to export data as a base for a developer survey.

10.1 Introduction

the SubCat methodology provides a rich framework for data extraction and mining convenience functionality. Part of the basic mining functionality of the SubCat implementation were used for the study presented in the next sections. Over the years, several studies have been performed on automatic classification of repository artefacts, be it bug reports (e.g. Antionol et al. in [88]) or commit messages (e.g. [96], [11], [95], [37]). However, most of these studies face the challenge of evaluating the performance of the approach internally and externally. A commonly used approach to measure success of a classification mechanism is to evaluate precision and recall or related values. These measurements are often performed internally by the researchers

themselves (e.g. see [94], [37] or [95]), or externally, by experts (see e.g. [11]). In both cases the evaluators are commonly not the authors of the changes and therefore cannot be sure of the intent of the change. Only the author of a change knows the desired effect of a commit.

Hence, our goal is to perform a survey on open source project contributors and have them classify their own changes by applying three different classification methods. We chose three classification schemes for our survey – more schemes would have reduced the number of classified commits, due to the time the developers were willing to invest. The first scheme is one implemented in e.g. [11] and is based on a slightly modified set of Swanson's maintenance task categories. The second scheme has been implemented by Hindle et al. in [96] and provides more detailed information on change intent by using non-functional requirements to classify commits. The third scheme is similar to the first scheme, but has been tailored specifically for software evolution in open source projects and was provided by Hattori and Lanza in [95]. Based on this data, future classification techniques might be evaluated using the gathered commit meta-information and existing approaches may be a posteriori re-evaluated for effectiveness.

10.2 Assembly of the data

To provide transparency on the process of data aggregation and transformation, the following section describes how the data for the final data set was assembled. We performed a four-steps process:

1. Selection of developers and projects with personal commitment
2. Assembly of commit data and creation of survey forms
3. Provisioning instructions and guidance
4. Aggregation of the data into a single data source

Selection of developers and projects

To perform the survey, we contacted developers that were or are regular committers to open source projects. All developers are senior developers (6+ years experience in open source or industry projects) and personal acquaintances of the authors of the survey. Due to the exhaustive scale of the survey, personal commitment of the developers was important, since the task could take more than two hours depending on the number of commits to be categorized. Seven developers including one of the authors of the study agreed to perform the categorization.

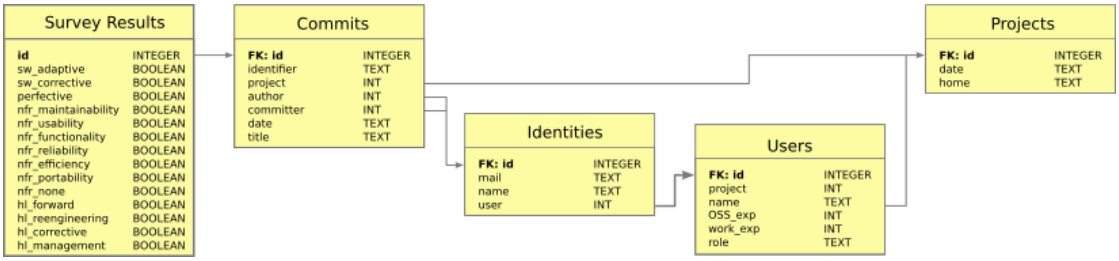


Figure 10.1: Data-Set Model for Multi-Projects Task Classification

The projects involved are very diverse, which may be beneficial for later research based on this data (e.g. in evaluation of cross-domain valid approaches). For example, Vala[1] is a compiler in the GNOME context, while Mylyn Reviews[2] is an established review tool provided as an Eclipse plug-in. The list of the projects may be found in the data set.

Assembly of commit data and creation of survey forms

We used the mining framework of the current implementation of the SubCat methodology to aggregate all data necessary for the survey forms. The framework offers, among other mining capabilities, functionality to mine GIT repositories and stores the mined information into an SQLite[3] database, which allowed for fast data exports into our survey forms. Every participant received a survey form that contained all of his commits from his open source project. The survey form supplied columns for each of the classification schemes as well as commit-identifiers, the commit message and meta-data on the change (e.g. changed files). We kept the initially generated database and stripped the model of all entities and attributes that were not relevant for the survey to be able to easily integrate the returned survey forms into the final data set. We inserted a table that contains the survey results into the SQLite database. This ensures comfortable integration of the data into any future studies. Figure 10.1 shows the model for the survey data.

Provide instructions and guidance

Adequate instructions as well as guidance provided to the developers who participated in the survey are crucial for the data quality in the final data set. Since classification is often ambiguous, we tried to provide the same basis for all participants. First, we provided a set of instructions for every participant. Once the participants read these, we supplied them with the survey sheet. Afterwards, an author of the study would make an appointment for a guidance session with the

[1] `https://wiki.gnome.org/Projects/Vala`
[2] `https://projects.eclipse.org/projects/mylyn.reviews`
[3] `http://www.sqlite.org/`

participant. In this session the participant could ask questions about the classification schemes and ask for assistance on the first set of commits. These sessions were intentionally held open, so that the author would not influence the participant in his application of the scheme, i.e. the author would only recount the instructions if the participant obviously misunderstood a category. Naturally, some questions on the schemes arose which may be found in section 10.3.

For the survey, we allowed for multi-labeling approaches, similarly to Hindle et al. in [96], since the primary goal of the survey is to capture as much of the intent of a change as possible. This is reflected in the instructions for all three schemes. We asked the participants to remove any changes where they were unsure or forgot about the intent of the change to increase the precision of the data set.

Instructions for Swanson's maintenance tasks

The first applied classification scheme is based on Swanson's maintenance tasks. It was applied in several studies (see e.g. [11]). We provided the participants with the following definitions of maintenance tasks (note that these definitions are altered from the original by Swanson to cover the open source development life cycle):

- **Corrective tasks** are corrective measures to the code base that address errors, faults or failures (according to the IEEE glossary for software engineering [50]) in the code base. This includes preventive maintenance steps taken to address latent faults that are not operational faults yet. E.g. a regression bug that is found during testing after the software was released. This does not include preventive measures that improve non-functional attributes.

- **Adaptive tasks** are changes that affect the business logic of the system, e.g. changes to implement new or alter existing functional requirements. These changes may stem from changes to the model of the software, but also from alterations of algorithms.

- **Perfective tasks** are enhancements to non-functional attributes of the system, e.g. boosting performance, refactoring or improving system documentation.

We asked the developers to classify all their commits according to this scheme. If a commit addressed a functional requirement, the developer was supposed to mark the commit as adaptive. They may assign multiple categories, if they thought a commit has addressed more than one category, e.g. a bug fix that also included some refactoring work would be marked as corrective and perfective. If all three categories were not fitting (e.g. actions like version tagging), we asked them to simply not mark the commit.

Instructions for NFR labeling The second classification of commits involves the non-functional requirements (NFR) a commit addresses. This classification is based on the ISO9126

quality model and was proposed by Hindle et al. in [96]. A commit may possibly be assigned to multiple NFRs - a commit may implement new features like a search bar, along with a design that improves usability of the product. Similarly to the previous classification, we wanted developers to assign the commit to every NFR that they thought applied to their change. If they found all six categories not fitting, they were supposed to mark the commit in the *none* column. The possible categories were:

- **Functionality:** The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. This includes *Suitability, Accuracy, Interoperability, Security and Functionality Compliance*.
- **Reliability:** The capability of the software product to maintain a specified level of performance when used under specified conditions. This includes *Maturity, Fault Tolerance, Recoverability and Reliability Compliance*.
- **Usability:** The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. This includes *Understandability, Learnability, Operability, Attractiveness and Usability Compliance*.
- **Efficiency:** The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. This includes *Time Behaviour, Resource Utilization and Efficiency Compliance*.
- **Maintainability:** The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. This includes *Analyzability, Changeability, Stability, Testability and Maintainability Compliance*.
- **Portability:** The capability of the software product to be transferred from one environment to another. This includes *Adaptability, Installability, Co-Existence, Replaceability and Portability Compliance*.

Instructions for software evolution tasks The third classification of commits is the scheme for activities during software evolution in open source projects, presented by Hattori and Lanza in [95]. The same procedure as with the initial classification was followed. We asked the developers to classify all of their commits according to this scheme. If a commit involved a forward engineering activity, they were supposed to mark it as forward engineering. Again, it was possible to assign multiple categories, if they thought a commit addressed more than one activity, e.g. a bug fix that also included some refactoring would be marked as corrective engineering and

re-engineering. If all four categories were not fitting, the developers did not mark them. Hattori and Lanza defined the following activities:

- **Forward engineering** activities are those related to incorporation of new features and implementation of new requirements.
- **Re-engineering** activities are related to refactoring, redesign and other actions to enhance the quality of the code without properly adding new features.
- **Corrective engineering** handles defects, errors and bugs in the software.
- **Management** activities are those unrelated to codification, such as formatting code, cleaning up, and updating documentation.

10.3 Results and discussion

Table 10.1 shows an overview of the returned results. For each change that was assigned to a category in a schema, we added one point to the overall category.

Table 10.1: Overview for Classification of Tasks by Developer

		Swanson's Tasks (%)			**NFR Labeling** (%)							**SW Evolution Tasks** (%)			
	total	*adap.*	*corr.*	*perf.*	*func.*	*maint.*	*usab.*	*rel.*	*eff.*	*port.*	*none*	*forw.*	*corr.*	*reeng.*	*man.*
Florian Brosch[a]	200	51,98	29,7	18,32	60,08	7,98	22,05	6,84	0,0	0,76	2,28	57,64	29,56	8,37	4,43
Luca Bruno[b]	116	37,29	54,24	8,47	66,93	6,3	9,45	3,15	3,94	0,79	9,45	34,75	50,85	10,17	4,24
Evan Nemerson[b]	194	34,36	59,49	6,15	51,59	5,1	41,4	0,32	0,0	0,64	0,96	32,67	56,44	4,46	6,44
Thomas Seidl[c]	118	18,52	55,56	25,93	26,35	7,43	20,95	14,86	4,73	3,38	22,3	12,3	48,36	9,84	29,51
Martin Reiterer[d]	123	19,82	19,82	60,36	10,42	44,44	3,47	6,25	3,47	8,33	23,61	11,97	16,2	34,51	37,32
Kilian Matt[e]	81	34,83	15,73	49,44	25,71	20,95	19,05	9,52	1,9	2,86	20,0	28,87	15,46	16,49	39,18
Mark Struberg[f]	135	15,6	41,13	43,26	28,11	10,14	13,82	4,15	0,92	19,82	23,04	15,65	34,01	16,33	34,01

[a] Valadoc, `http://valadoc.org/`, 50.709 LoC
[b] Vala, `http://vala-project.org/`, 236.071 LoC
[c] Drupal Search API, `http://drupal.org/project/search_api`, 21.696 LoC
[d] TapiJI, `http://code.google.com/a/eclipselabs.org/p/tapiji/`, 19.611 LoC
[e] MyLyn, `http://eclipse.org/mylyn/`, 76.464 LoC
[f] DeltaSpike, `http://deltaspike.apache.org/`, 35.202 LoC

A study by Godfrey and Tu [52] suggested that preventive maintenance and planned evolution was playing only a minor role in open source software evolution. Just by looking at the large number of perfective changes coupled with specific development roles in our survey, perfective changes and thus planned evolution as well as preventive maintenance seem to be more than a minor factor in software evolution. We suggest to perform interviews with the developers with a high perfective change profile to learn more about the nature of the perfective changes.

Aggregation of the data into a single data source

Before we aggregated the data, we removed the first ten commits classified by the developer from each survey form. This was necessary, since the authors assisted during some of these commits and feedback form the developers suggested that ten commits is a sufficiently large training set to understand each classification scheme.

Once we removed the first ten commits, we aggregated the survey forms and imported them into the database. The database holds the raw data in the table `Commits` and the manually classified commits in the table `SurveyResults`. It also holds a table `Identitites` to store user and author information. Furthermore it stores general project information in `Projects`. Figure 10.1 shows the model for the final data set. Table `Users` is necessary due to compatibility reasons with the framework implementation so BTS account information may be mapped to commit messages in future studies.

Discussion on the classification schemes

Feedback from the developers showed that all of the classification schemes were easy to apply and were perceived as coherent. Some problems occurred when changes did not fit any of the options provided by a scheme, e.g. developers reported that the category 'Management' of Hattori and Lanza [95] proved useful to identify changes they performed to tag versions or to create private branches. An example for this was the commit with the message "Creating private branch for the implementation of key refactoring functionality." which could be labeled with the third categorization scheme, while it could only be categorized as "None" in the second scheme and could not be categorized at all according to the first scheme.

One of the developers (Evan Nemerson) in the survey turned out to perform a specialized role in the Vala project. He is responsible for maintaining language bindings for Vala. His tasks are either triggered by an issue report, or by an update in a dependent API. Hence his task spectrum should be very one dimensional in all three schemes. This might be interesting for further research on developer roles.

Luca Bruno from the Vala project pointed out that a lot of his development tasks are also based on bug reports. Similarly to Evan, bug reports trigger his development work – sometimes his commit ends up including a bug fix as well as major new functionality. This means that the boundary between a corrective maintenance task and an adaptive maintenance task (analogously a corrective engineering activity and a forward engineering activity) are not clearly defined in some cases. The second classification scheme avoids this issue, since bug-fixing and adding functionality both fall into the *Functionality* category.

A discussion that was brought up by Florian Brosch during his classification run was the ambiguity of *Usability* improvements. Even he, as a co-author of this paper, had troubles to determine whether a change was solely *perfective* or *adaptive* or both. The second scheme almost always resulted in *Functionality* and *Usability*. The third scheme is difficult as well, since it may be constructed that a *Usability* framework may be implemented without any features. In this case it might be considered *forward-engineering*, however if previously features existed, it may be argued that the same feature could now be considered *re-engineering*.

10.4 Conclusion

The data set we provide contains 967 classified commits. Every commit has been enriched with meta-information whether it addresses functionality requirements or non-functional requirements. Further, a fine-grained mapping for non-functional requirements addressed by a change is provided, as well as whether the change was refactoring related or a repository management elicited change. The provided meta-information allows to evaluate existing labeling and classification approaches, be it machine learners or otherwise generated schemes. More importantly though, the information may be used to train new classification techniques, since all necessary information to match the survey to the existing project repositories is available. Even more complex approaches that leverage not only code repository information but also e.g. bug tracking systems may be applied, due to the available author information.

The data is provided as an SQLite database and can easily be imported or statistically analyzed (e.g. by using RSQLite[4]). The model of the data is simple and intuitive and may easily be integrated into existing approaches that leverage more than one repository type (e.g. by using the user table to match code repository and bug tracker system users). Since one of the authors of the paper classified his own changes, we suggest to treat his data set differently, since his more thorough knowledge of the classification mechanisms might introduce bias.

As for the classification schemes, developer acceptance was high. Developers did not require further explanation of the schemes beyond the initial instructions and the guidance session.

[4] `http://cran.r-project.org/web/packages/RSQLite/index.html`

Three preferred to step together through the first changes, while the rest did not. Overall though, feedback on all three schemes was that the schemes felt coherent and applicable.

We successfully demonstrated the use of SubCat functionality to provide the initial data for this survey, thus showing another possible use scenario for software evolution research.

Part IV

Outcome

CHAPTER 11

Conclusion

Da steh' ich nun, ich armer Tor,
und bin so klug als wie zuvor!

Faust, Teil 1 [124]

Contrary to Dr. Faust in the introductory quote, technically (albeit not philosophically) speaking, the introduced methodology has generated new insights in software evolution research. We presented a novel approach for a top-down analysis based on natural language processing capabilities. In chapter 1 we proposed the following contributions of this thesis:

- The SubCat methodology to gather information and provide insights into software maintenance processes
- An implementation of the methodology as a java-based tool
- The application of the methodology

11.1 The SubCat Methodology

The SubCat methodology as presented in this thesis focused on aspects that current methods and frameworks in software evolution neglect or have shortcomings in. These aspects have been pointed out in chapter 2 and have been addressed in the design of the methodology, the architecture of its implementation and the entity design. There are three key aspects that the SubCat methodology provides improvements in. These are **integration of data sources**, **transportability** and **applicability**.

Integration of various data sources on software evolution has been identified as a central attribute for analysis of software evolution in existing literature. The presented methodology incorporates this assessment into its design and proves its importance based on the following considerations.

- In chapter 8, the proposed methodology is implemented to integrate BTS and VCS data. We show that many corrective maintenance tasks are not tracked in or linked to the BTS and are therefore not considered outside of the SubCat methodology (e.g. in [125] or in [90] and many more). This has also been acknowledged as an issue by Herzig et al. in [126]

- In chapter 7, the proposed methodology is implemented to analyze security issues. As external data source, the FreeBSD advisories and their retrospective IDs are used to validate the classified security changes in the VCS

- SubCat proposes the theoretical use of techniques like sentiment analysis and classification of software artifacts (see chapter 3). These techniques may be used on any body of natural language text, therefore integration of various data sources is mandatory and has been achieved by the current implementation (see chapter 4).

Transportability of the methodology is one aspect of the low acceptance of tooling and framework solutions in current software evolution research. This has been addressed by designing SubCat based on natural language processing faculties and by using naive approaches over sophisticated complex ones. The efficacy of this has been proven by the following findings:

- Natural language is the common denominator between different types of software repositories, hence methodologies based on natural language may be used on every type of software project, including domain or programming language. Applications of SubCat on many projects in different contexts in section III prove the usefulness of this approach

- In chapter 6 we show that a naive approach provides more favorable results for change classification in software evolution than complex or automated approaches based on machine learners as presented e.g. in [96]

- Chapter 6 and chapter 7 shows the importance of transportability between domains. We prove that the SubCat methodology may be applied for security change analysis as well as maintenance task classification

- In chapter 6 we also prove that naive approaches are transportable without further configurations and may be used for different projects. We validate our findings in a developer survey

Applicability of the approach is, next to transportability, the other major shortcoming of existing frameworks and tools. While there exist many methods for analysis on software evolution, they commonly lack applicability and need to be tuned for a specific project. Furthermore, their output is targeted at researchers mostly.

- In chapter 4 we show the various possible output forms of SubCat, while chapter 9 holds a practical application of these output forms to allow e.g. a quick overview of developer motivation over time
- SubCat's methodology includes a generic, component-based architecture and a robust data model, which is shown in chapter 3, making it easy to understand, set-up and apply
- Applicability is closely tied to compliance with non-functional requirements like performance. The three-tiered architecture supports this notion by de-coupling processes and process steps according to a proposed, best-practice mining process (see [6]). This is described in detail in chapter 3

11.2 Implementation of the SubCat Methodology

To achieve the goal of a robust framework that addresses issues of applicability and transportability, it is necessary to use a stable process that has a general consensus as a best practice in the research community and a concept of entities that is theoretically capable of covering the most commonly known data sources in software evolution research. Further, it is required that the implementation is extensible to include new data sources and analysis techniques (customized or standard libraries). Since applicability is dependent on non-functional requirements like performance, the design of the implementation must address these issues. Figure 11.1 contains the resulting layered architecture to provide the desired features.

To address the three major aspects listed in the previous section, the implementation of the SubCat methodology delivers the following features:

- Realization of a best-practice mining process described in chapter 3 and 4 to provide insights into software evolution for both researchers and practitioners

- Persistence of data in a generic data model that is flexible and may be extended to include various other data sources, which is described theoretically in chapter 3 and applied in chapter 8 and 10.
- Compartmentalized functionality that may easily be expanded to include new data sources or new analysis techniques, shown in chapter 8 and 7
- Adherence to non-functional requirements according to findings in literature and personal experience as described in chapter 4 to generate an overall enjoyable user experience

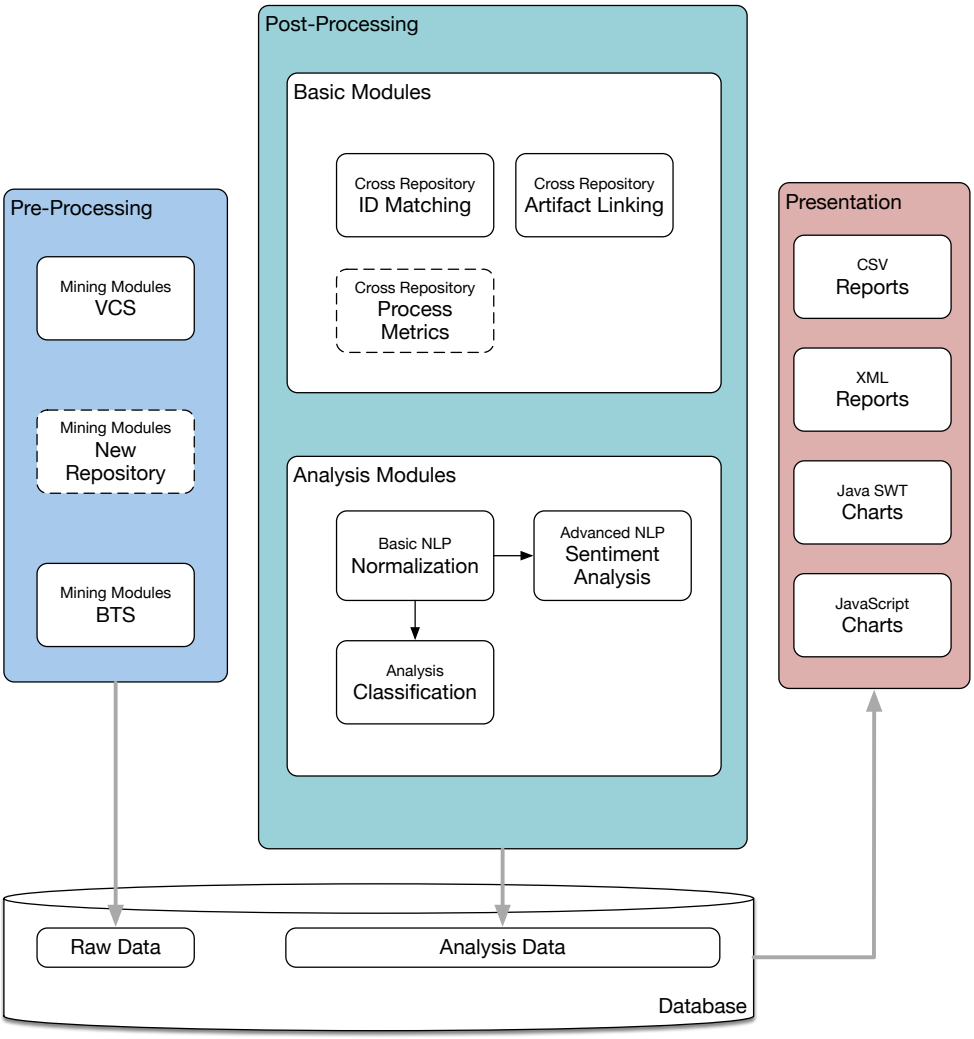


Figure 11.1:
SubCat Architecture - Overview

11.3 Application of the Methodology

By applying the SubCat methodology in various contexts we could report a number of findings, which included (among others), that security critical modules may be found automatically by applying a dictionary of security related terms, that (unsurprisingly) BTS do not offer any ground truth on the reality of bugs in a code base or that a manually generated dictionary out-performs machine learners, especially so for cross-project validity. The design of the SubCat methodology and its application in the conducted studies in consequence has been based on a top down approach on processing natural language, i.e. the assumption that natural language expresses intent more coherently than source code. This assumption has been validated in the conducted studies - the bodies of text accompanying changes to software always delivered more or less precise information on the intent of the change. The presented methodology has been applied in the following studies and surveys that have proven the efficacy and usefulness of the approach:

- Applying the SubCat methodology for a preliminary feasibility study
  - The SubCat methodology is initially employed in a preliminary study on **correlation between classified change** and a set of **project metrics**. The aim of the study was to showcase the potential of using an extended approach based on the initial setup presented in Mockus and Hassan's work.
- Applying the SubCat methodology for security analysis:
  - The flexibility of the SubCat methodology was used **to identify security related changes** to modules in FreeBSD. This allowed to identify security critical components of FreeBSD and map them against known vulnerabilities and security issues
- Applying the SubCat methodology for change classification
  - SubCat was used to create and **validate a cross-project valid dictionary** that outperformed existing classification tools in precision and recall. The dictionary was evaluated in a developer survey and showed moderate agreement with developer opinion based on changed source code and commit messages
- Applying the SubCat methodology to populate an issue tracker
  - A new java command line tool was implemented to show that due to its design the SubCat methodology may be integrated with a new tool to **populate a bug database** from a VCS

- Applying the SubCat methodology to create developer profiles:
  - SubCat was used to perform sentiment analysis and change classification to **create developer profiles** in open source projects
- Applying the SubCat methodology in a survey for commit classification:
  - The output of the implementation of SubCat was used for a **ground truth evaluation** on change classification with the original commit authors of various open source initiatives

CHAPTER 12

Future Works

There are several steps in the further development of the SubCat methodology. These steps can be divided into short, mid and long term changes or improvements to the methodology. Short term changes concern additional features to the current implementation of the methodology. Mid term changes concern the overall evolution of the architecture of SubCat, while long term changes take modifications to the overall methodology into consideration.

12.1 Short Term Modifications and Improvements

The immediate changes to the current SubCat implementation concern the integration of new specific repository technologies into the BTS and VCS modules. According to the popular open source software development indexing site `http://openhub.net`, which indexed over 600.000 projects in January 2016, the most popular VCS[1] are Subversion (47%), GIT (39%), CVS (9%) followed by Mercurial (2%) and Bazaar (1%). Since CVS is constantly declining and has shifted out of focus of active projects, the next possible integration interface for the current SubCat implementation should be Mercurial and/or Bazaar, so further projects could be analyzed.

Unfortunately, there is no data on issue tracker popularity readily available. There are however at least two specific BTS that would be convenient to integrate into the current SubCat implementation. The first is JIRA, an issue tracker with a wide industry acceptance and which has been used in various projects by the author. Integration of JIRA would allow to make use of anonymized, available industrial project data. The second option of extension is the GitHub

[1] `https://www.openhub.net/repositories/compare` - last visited on 24.06. 2016

issue tracker. Benefits of this implementation are obvious, as GitHub hosts a broad range of freely accessible projects that could be analyzed.

Another aspect that could potentially be expanded are output channels in the SubCate implementation. Currently, SubCat delivers outputs via an SWT-based front-end, as reports in comma-separated value format and rendered as a web-page to be integrated into build pipelines. The next step of improvement here would be to use a report-generating library like BIRT[2] to integrate these various output channels. However, the learning curve for BIRT is steep and the framework offers a broad range of features. The implementation of a reporting/visualization platform like BIRT seems a sensible extension to SubCat, but would need a detailed cost-benefit analysis.

One software engineering best practice has been currently neglected in the implementation of SubCat and should be added as soon as possible. SubCats current implementation lacks test automation at the system test level. It would be beneficial to create an automation framework, since the broad acceptance and usage of SubCat is one of the major goals of SubCat. High product quality is one condition to reach this goal.

12.2 Mid Term Modifications and Improvements

The mid term goals for the SubCat methodology concern the architecture of SubCat. Further types of repositories may be integrated to provide additional data. The general facilities to link accounts between various repositories exists, so do pre-processing features like the removal of technical information from bodies of text. Further, sentiment analysis and classification mechanisms could easily be applied on any other type of repository. Since the data model was designed with extensibility in mind, only slight modifications in the data model would be needed. Most changes would happen in the application layer in new modules. Interesting repository types could include e.g. mailing lists or IRC logs.

SubCat has been implemented in a first iteration using WordNet[3] to provide NLP capabilities. Recently, Stanford NLP[4] has replaced WordNet, for usability and performance reasons. However, SentiWordNet[5] is still used for sentiment analysis. There might be alternatives accessible that should be taken into consideration. Also, the used libraries could be upgraded to more recent versions (e.g. java, SQLite).

[2] http://www.eclipse.org/birt/
[3] http://wordnet.princeton.edu/
[4] http://nlp.stanford.edu/
[5] http://sentiwordnet.isti.cnr.it/

12.3 Long Term Modifications and Improvements

In the long run, the SubCat methodology itself has room for potential improvement. The pre-processing capabilities generate a data base which might be used for any advanced analysis library like machine learners. So it would make sense to extend the SubCat methodology as a whole to integrate machine learners and to use the output channels of SubCat to provide the analyzed data. This would considerably increase the potential of correlating analysis results easily. However, this would also decrease usability of the system as a whole, so the main argument for the SubCat methodology, applicability and transportability needs to be re-evaluated against the possible benefits of integrating new, complex analysis methods into the tooling landscape.

12.4 Possible Applications of SubCat

Aside modifications and improvements, there are also many possible applications for the SubCat methodology. One of these applications is the analysis of motivational factors based on task profiles and sentiment expressed in software artifacts. Another possible application would be sentiment expressed at a certain time in a project, e.g. is it possible to see an approaching release date based on the language used in a project. Further applications of SubCat could be found by using specialized dictionaries - there are approaches in literature that use different labeling categories, e.g. Hindle's work on non-functional requirements mapping (see [96]). It would possibly be beneficial to repeat the study presented in chapter 7 on more than one project, to see if the approach holds merit beyond the scope of FreeBSD. Furthermore, it would be interesting to analyze the implications of composite code changes (see Yida Tao and Sunghun Kim in [127]) on classification techniques that use comments to classify these and compare the results to code based classification techniques.

Bibliography

[1] International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering - Software Life Cycle Processes - Maintenance 2006, doi:10.1109/IEEESTD.2006.235774.

[2] Bennett KH, Rajlich VT. Software maintenance and evolution. *Proceedings of the conference on The future of Software engineering - ICSE '00*, ACM Press: New York, New York, USA, 2000; 73–87, doi:10.1145/336512.336534. URL `http://dl.acm.org/citation.cfm?id=336512.336534`.

[3] Mockus A, Fielding RT, Herbsleb JD. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* Jul 2002; **11**(3):309–346, doi:10.1145/567793.567795. URL `http://portal.acm.org/citation.cfm?doid=567793.567795`.

[4] Bernhart M, Mauczka A, Fiedler M, Strobl S, Grechenig T. Incremental reengineering and migration of a 40 year old airport operations system. *IEEE International Conference on Software Maintenance, ICSM*, 2012; 503–510.

[5] Lin YR, Sundaram H, Chi Y, Tatemura J, Tseng BL. Blog Community Discovery and Evolution Based on Mutual Awareness Expansion. *IEEE/WIC/ACM International Conference on Web Intelligence (WI'07)*, IEEE, 2007; 48–56, doi:10.1109/WI.2007.71. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4427065`.

[6] Hemmati H, Nadi S, Baysal O, Kononenko O, Wang W, Holmes R, Godfrey MW. The MSR Cookbook: Mining a decade of research. *2013 10th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2013; 343–352, doi:10.1109/MSR.2013.6624048. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6624048`.

[7] Gómez R. Exploring Expressions of Emotions in GitHub Commit Messages. May 2012; URL `http://www.webcitation.org/6N9nD4IQN`.

[8] Brosch FV. *Mining software repositories; implementing a light-weight tool in the MSR area by applying NLP methods*. 2015. Zusammenfassung in deutscher Sprache; Diplomarbeit Technische Universität Wien 2015.

[9] Bevan J, Whitehead EJ, Kim S, Godfrey M. Facilitating software evolution research with kenyon. *ACM SIGSOFT Software Engineering Notes* Sep 2005; **30**(5):177, doi: 10.1145/1095430.1081736. URL `http://portal.acm.org/citation.cfm?doid=1095430.1081736`.

[10] Mauczka A, Bernhart M, Grechenig T. Analyzing the relationship of process metrics and classified changes - a pilot study. *SEKE*, 2010; 269–272.

[11] Mauczka A, Huber M, Schanes C, Schramm W, Bernhart M, Grechenig T. Tracing your maintenance work–a cross-project validation of an automated classification dictionary for commit messages. *Fundamental Approaches to Software Engineering*. Springer, 2012; 301–315.

[12] Swanson EB. The dimensions of maintenance. *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, 1976; 492–497.

[13] Zvegintzov N. Nanotrends. *Datamation 29.8* 1983; :106–108.

[14] Schneidewind N. The State of Software Maintenance. *IEEE Transactions on Software Engineering* Mar 1987; **SE-13**(3):303–310, doi:10.1109/TSE.1987.233161. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702216`.

[15] Erlikh L. Leveraging legacy system dollars for e-business. *IT Professional* 2000; **2**(3):17–23, doi:10.1109/6294.846201. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=846201`.

[16] Raymond E. The cathedral and the bazaar. *Knowledge, Technology & Policy* 1999; **12**(3):23–49, doi:10.1007/s12130-999-1026-0. URL `http://dx.doi.org/10.1007/s12130-999-1026-0`.

[17] Sutherland JV, Schwaber K. The scrum methodology. *Business object design and implementation: OOPSLA workshop*, 1995.

[18] Mockus, Votta. Identifying reasons for software changes using historic databases. *Proceedings of the International Conference on Software Engineering* 2000; :120–130.

[19] Hassan AE. Automated classification of change messages in open source projects. *Proceedings of the 2008 ACM symposium on Applied computing - SAC '08* 2008; :837doi:10.1145/1363686.1363876. URL http://portal.acm.org/citation.cfm?doid=1363686.1363876.

[20] Śliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories* Jul 2005; URL http://portal.acm.org/citation.cfm?id=1083142.1083147.

[21] Neuhaus S, Zimmermann T, Holler C, Zeller A. Predicting vulnerable software components. *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* Oct 2007; URL http://portal.acm.org/citation.cfm?id=1315245.1315311.

[22] Zimmermann T. Fine-grained processing of cvs archives with apfel. *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange* Oct 2006; URL http://portal.acm.org/citation.cfm?id=1188835.1188839.

[23] Rigby P, Hassan A. What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list. *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on* Apr 2007; :23 – 23doi:10.1109/MSR.2007.35. URL http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4228660&isnumber=4228635&punumber=4228634&k2dockey=4228660@ieeecnfs.

[24] Hassan AE. Predicting faults using the complexity of code changes. *2009 IEEE 31st International Conference on Software Engineering* 2009; :78–88doi:10.1109/ICSE.2009.5070510. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5070510.

[25] Fischer M, Pinzger M, Gall H. Populating a Release History Database from version control and bug tracking systems. *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* 2003; :23–32doi:10.1109/ICSM.2003.1235403. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1235403.

[26] Abebe SL, Haiduc S, Marcus A, Tonella P, Antoniol G. Analyzing the Evolution of the Source Code Vocabulary. *2009 13th European Conference on Software Maintenance and Reengineering* 2009; :189–198doi:10.1109/CSMR.2009.

61. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4812752.

[27] Fluri B, Wursch M, Gall HC. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. *14th Working Conference on Reverse Engineering (WCRE 2007)* Oct 2007; :70–79doi:10.1109/WCRE.2007.21. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4400153.

[28] Becca ML. Clustering Support for Fault Prediction in Software. *International Conference on Software Maintenance, 2012. ICSM 2012. Proceedings.* 2012; .

[29] Hindle A, Barr ET, Su Z, Gabel M, Devanbu P. On the naturalness of software. *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012; 837–847, doi:10.1109/ICSE.2012.6227135. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6227135.

[30] Murgia A, Tourani P, Adams B, Ortu M. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, ACM Press: New York, New York, USA, 2014; 262–271, doi:10.1145/2597073.2597086. URL http://dl.acm.org/citation.cfm?id=2597073.2597086.

[31] Guzman E, Bruegge B. Towards emotional awareness in software development teams. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, ACM Press: New York, New York, USA, 2013; 671, doi:10.1145/2491411.2494578. URL http://dl.acm.org/citation.cfm?id=2491411.2494578.

[32] Pinzger M, Gall H, Jazayeri M. ArchView - Analyzing Evolutionary Aspecs of Complex Software Systems. *Vienna University of Technology* 2005; URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.212&rep=rep1&type=pdf.

[33] Burch M, Diehl S, Weibgerber P. EPOSee — A Tool For Visualizing Software Evolution. *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis* 2005; :1–2doi:10.1109/VIssOF.2005.1684322. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1684322.

[34] Gluetheos: automating the retrieval and analysis of data from publicly available software repositories. *IET Conference Proceedings* January 2004; :28–31(3)URL http://digital-library.theiet.org/content/conferences/10.1049/ic_20040471.

[35] Zimmermann T. Fine-grained processing of CVS archives with APFEL. *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange - eclipse '06* 2006; :16–20doi:10.1145/1188835.1188839. URL http://portal.acm.org/citation.cfm?doid=1188835.1188839.

[36] Zimmermann T, Weiß gerber P, Diehl S, Zeller A. Mining Version Histories to Guide Software Changes 2004; .

[37] Mockus A, Votta L. Identifying reasons for software changes using historic databases. *Software Maintenance, 2000. Proceedings. International Conference on* Sep 2000; :120 – 130doi:10.1109/ICSM.2000.883028. URL http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=883028&isnumber=19102&punumber=7091&k2dockey=883028@ieeecnfs.

[38] Mauczka A, Schanes C, Fankhauser F, Bernhart M, Grechenig T. Mining security changes in freebsd. *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories* 2010; :90–93.

[39] Mauczka A, Brosch F, Schanes C, Grechenig T. Dataset of Developer-Labeled Commit Messages. *2015 12th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2015.

[40] Brosch F, Mauczka A, Grechenig T. Technical report: Implementing a light-weight tool in the msr area by applying nlp methods. *Technical Report, 2015* 2016; .

[41] Wagner T, Mauczka A. Technical report: Populating a bug database by using repositories. *Technical Report, 2015* 2015; .

[42] Paine A. *Mark Twain: A Biography*. v. 2, BiblioBazaar, 2008. URL http://books.google.at/books?id=5vUeaTrA8QUC.

[43] Beck K. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[44] Koch SVUoE, BA). *Extreme Programming and Agile Processes in Software Engineering*, *Lecture Notes in Computer Science*, vol. 3092. Springer Berlin Heidelberg: Berlin,

Heidelberg, 2004, doi:10.1007/b98150. URL http://www.springerlink.com/index/10.1007/b98150.

[45] Aversano L, Canfora G, Cimitile A, De Lucia A. Migrating legacy systems to the Web: an experience report. *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, IEEE Comput. Soc, 2001; 148–157, doi:10.1109/.2001.914979. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=914979.

[46] Lehmann MM. Life cycles and laws of software evolution. *Proceedings of th IEEE* 1980; **68**:1060–1076.

[47] Lehman M, Ramil J, Wernick P, Perry D, Turski W. Metrics and laws of software evolution-the nineties view. *Proceedings Fourth International Software Metrics Symposium*, IEEE Comput. Soc, 1997; 20–32, doi:10.1109/METRIC.1997.637156. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=637156.

[48] Lientz BP, Swanson EB. Problems in application software maintenance. *Communications of the ACM* Nov 1981; **24**(11):763–769, doi:10.1145/358790.358796. URL http://dl.acm.org/citation.cfm?id=358790.358796.

[49] Godfrey MW, German DM. The past, present, and future of software evolution. *2008 Frontiers of Software Maintenance*, IEEE, 2008; 129–138, doi:10.1109/FOSM.2008.4659256. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4659256.

[50] IEEE Standard Glossary of Software Engineering Terminology 1990, doi:10.1109/IEEESTD.1990.101064.

[51] Koponen T, Hotti V. Open source software maintenance process framework. *ACM SIGSOFT Software Engineering Notes* Jul 2005; **30**(4):1, doi:10.1145/1082983.1083265. URL http://dl.acm.org/citation.cfm?id=1082983.1083265.

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

[53] Reifer D. *Software Maintenance Success Recipes*. Taylor & Francis, 2012. URL http://books.google.com/books?id=N7vCFAadRq4C.

[54] Aberdour M. Achieving Quality in Open-Source Software. *IEEE Software* jan 2007; **24**(1):58–64, doi:10.1109/MS.2007.2. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4052554.

[55] Rigby PC, German DM, Cowen L, Storey MA. Peer review on open source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology* 2014; :34.

[56] Begel A, Nagappan N. Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, IEEE, 2007; 255–264, doi:10.1109/ESEM.2007.12. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4343753.

[57] Talby D, Dubinsky Y. Governance of an agile software project. *2009 ICSE Workshop on Software Development Governance*, IEEE, 2009; 40–45, doi:10.1109/SDG.2009.5071336. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5071336.

[58] Rodríguez P, Markkula J, Oivo M, Turula K. Survey on agile and lean usage in finnish software industry. *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*, ACM Press: New York, New York, USA, 2012; 139, doi:10.1145/2372251.2372275. URL http://dl.acm.org/citation.cfm?id=2372251.2372275.

[59] Mockus A, Fielding RT, Herbsleb J. A case study of open source software development. *Proceedings of the 22nd international conference on Software engineering - ICSE '00*, ACM Press: New York, New York, USA, 2000; 263–272, doi:10.1145/337180.337209. URL http://dl.acm.org/citation.cfm?id=337180.337209.

[60] Chen JC, Huang SJ. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software* 2009; **82**(6):981–992, doi:10.1016/j.jss.2008.12.036.

[61] Paulson J, Succi G, Eberlein A. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering* apr 2004; **30**(4):246–256, doi:10.1109/TSE.2004.1274044. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1274044.

[62] Hall T, Sharp H, Beecham S, Baddoo N, Robinson H. What Do We Know about Developer Motivation? *IEEE Software* jul 2008; **25**(4):92–94, doi:10.1109/MS.2008.105. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4548414.

[63] Hertel G, Niedner S, Herrmann S. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research policy* 2003; **32**(7):1159–1177.

[64] Fielding RT. Shared leadership in the Apache project. *Communications of the ACM* Apr 1999; **42**(4):42–43, doi:10.1145/299157.299167. URL http://dl.acm.org/ft_gateway.cfm?id=299167&type=html.

[65] Bird C, Pattison D, D'Souza R, Filkov V, Devanbu P. Latent social structure in open source projects. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, ACM Press: New York, New York, USA, 2008; 24, doi:10.1145/1453101.1453107. URL http://dl.acm.org/citation.cfm?id=1453101.1453107.

[66] Hong Q, Kim S, Cheung S, Bird C. Understanding a developer social network and its evolution. *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2011; 323–332, doi:10.1109/ICSM.2011.6080799. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6080799.

[67] Nia R, Bird C, Devanbu P, Filkov V. Validity of network analyses in Open Source Projects. *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, IEEE, 2010; 201–209, doi:10.1109/MSR.2010.5463342. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5463342.

[68] Tsay J, Dabbish L, Herbsleb J. Influence of social and technical factors for evaluating contribution in GitHub. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, ACM Press: New York, New York, USA, 2014; 356–366, doi:10.1145/2568225.2568315. URL http://dl.acm.org/citation.cfm?id=2568225.2568315.

[69] Crowston K, Wei K, Howison J, Wiggins A. Free/Libre open-source software development. *ACM Computing Surveys* Feb 2012; **44**(2):1–35, doi:10.1145/2089125.2089127. URL http://dl.acm.org/citation.cfm?id=2089125.2089127.

[70] Shah SK. Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development. *Management Science* Jul 2006; **52**(7):1000–1014, doi:10.1287/mnsc.1060.0553. URL http://dl.acm.org/citation.cfm?id=1246148.1246152.

[71] Wu CG, Gerlach JH, Young CE. An empirical analysis of open source software developers' motivations and continuance intentions. *Information & Management* Apr 2007; **44**(3):253–262, doi:10.1016/j.im.2006.12.006. URL http://dl.acm.org/citation.cfm?id=1243524.1243851.

[72] Fang Y, Neufeld D. Understanding Sustained Participation in Open Source Software Projects. *Journal of Management Information Systems* Apr 2009; **25**(4):9–50, doi:10.2753/MIS0742-1222250401. URL http://dl.acm.org/citation.cfm?id=1554441.1554443.

[73] Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P. The promises and perils of mining git. *2009 6th IEEE International Working Conference on Mining Software Repositories* May 2009; :1–10doi:10.1109/MSR.2009.5069475. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5069475.

[74] Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D. The promises and perils of mining GitHub. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, ACM Press: New York, New York, USA, 2014; 92–101, doi:10.1145/2597073.2597074. URL http://dl.acm.org/citation.cfm?id=2597073.2597074.

[75] Demeyer S, Murgia A, Wyckmans K, Lamkanfi A. Happy Birthday! A trend analysis on past MSR papers. *2013 10th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2013; 353–362, doi:10.1109/MSR.2013.6624049. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6624049.

[76] Hassan AE. The road ahead for Mining Software Repositories. *2008 Frontiers of Software Maintenance*, IEEE, 2008; 48–57, doi:10.1109/FOSM.2008.4659248. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4659248.

[77] Allamanis M, Sutton C. Mining source code repositories at massive scale using language modeling. *2013 10th Working Conference on Mining Software Repositories (MSR)*, IEEE,

2013; 207–216, doi:10.1109/MSR.2013.6624029. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6624029.

[78] Allamanis M, Sutton C. Why, when, and what: Analyzing Stack Overflow questions by topic, type, and code. *2013 10th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2013; 53–56, doi:10.1109/MSR.2013.6624004. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6624004.

[79] Guzzi A, Bacchelli A, Lanza M, Pinzger M, van Deursen A. Communication in open source software development mailing lists. *2013 10th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2013; 277–286, doi:10.1109/MSR.2013.6624039. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6624039.

[80] Kevic K, Fritz T. A dictionary to translate change tasks to source code. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, ACM Press: New York, New York, USA, 2014; 320–323, doi:10.1145/2597073.2597095. URL http://dl.acm.org/citation.cfm?id=2597073.2597095.

[81] Merten T, Mager B, Bürsner S, Paech B. Classifying unstructured data into natural language text and technical information. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, ACM Press: New York, New York, USA, 2014; 300–303, doi:10.1145/2597073.2597112. URL http://dl.acm.org/citation.cfm?id=2597073.2597112.

[82] Bacchelli A, D'Ambros M, Lanza M. Extracting Source Code from E-Mails. *2010 IEEE 18th International Conference on Program Comprehension*, IEEE, 2010; 24–33, doi:10.1109/ICPC.2010.47. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5521781.

[83] Bacchelli A, Cleve A, Lanza M, Mocci A. Extracting structured data from natural language documents with island parsing. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE, 2011; 476–479, doi:10.1109/ASE.2011.6100103. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6100103.

[84] Bettenburg N, Adams B, Hassan AE, Smidt M. A Lightweight Approach to Uncover Technical Artifacts in Unstructured Data. *2011 IEEE 19th International Conference on Program Comprehension*, IEEE, 2011; 185–188, doi:10.1109/ICPC.2011.

36. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5970153.

[85] Wang X, Zhang L, Xie T, Anvik J, Sun J. An approach to detecting duplicate bug reports using natural language and execution information. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, ACM Press: New York, New York, USA, 2008; 461, doi:10.1145/1368088.1368151. URL http://dl.acm.org/citation.cfm?id=1368088.1368151.

[86] Amor JJ, Robles G, Gonzalez-Barahona JM, Navarro A. Discriminating development activities in versioning systems: A case study. Citeseer.

[87] Hindle A, German DM, Godfrey MW, Holt RC. Automatic classication of large changes into maintenance categories. *2009 IEEE 17th International Conference on Program Comprehension*, IEEE, 2009; 30–39, doi:10.1109/ICPC.2009.5090025. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5090025.

[88] Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG. Is it a bug or an enhancement? *Proceedings of the 2008 conference of the center for advanced studies on collaborative research meeting of minds - CASCON '08*, ACM Press: New York, New York, USA, 2008; 304, doi:10.1145/1463788.1463819. URL http://dl.acm.org/citation.cfm?id=1463788.1463819.

[89] Kim S, Jr EJW, Zhang Y. Classifying Software Changes : Clean or Buggy ? 2008; **34**(2):181–196.

[90] Śliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes* Jul 2005; **30**(4):1, doi:10.1145/1082983.1083147. URL http://portal.acm.org/citation.cfm?doid=1082983.1083147.

[91] Kuhn A. Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code. *2009 6th IEEE International Working Conference on Mining Software Repositories* May 2009; :175–178doi:10.1109/MSR.2009.5069499. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5069499.

[92] De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S. Using IR methods for labeling source code artifacts: Is it worthwhile? *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, IEEE, 2012; 193–202, doi:10.1109/ICPC.2012.

6240488. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6240488.

[93] Blei DM, Ng AY, Jordan MI. Latent dirichlet allocation. *The Journal of Machine Learning Research* Mar 2003; **3**:993–1022. URL http://dl.acm.org/citation.cfm?id=944919.944937.

[94] Fu Y, Yan M, Zhang X, Xu L, Yang D, Kymer JD. Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation. *Information and Software Technology* Jun 2014; doi:10.1016/j.infsof.2014.05.017. URL http://www.sciencedirect.com/science/article/pii/S0950584914001347.

[95] Hattori LP, Lanza M. On the nature of commits. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, IEEE, 2008; 63–71, doi:10.1109/ASEW.2008.4686322. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4686322.

[96] Hindle A, Ernst NA, Godfrey MW, Mylopoulos J. Automated topic naming to support cross-project analysis of software maintenance activities. *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, ACM Press: New York, New York, USA, 2011; 163, doi:10.1145/1985441.1985466. URL http://dl.acm.org/citation.cfm?id=1985441.1985466.

[97] Henβ S, Monperrus M, Mezini M. Semi-automatically extracting FAQs to improve accessibility of software development knowledge Jun 2012; :793–803URL http://dl.acm.org/citation.cfm?id=2337223.2337317.

[98] Han D, Zhang C, Fan X, Hindle A, Wong K, Stroulia E. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. *2012 19th Working Conference on Reverse Engineering*, IEEE, 2012; 83–92, doi:10.1109/WCRE.2012.18. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6385104.

[99] Hindle A, Bird C, Zimmermann T, Nagappan N. Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers? *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2012; 243–252, doi:10.1109/ICSM.2012.6405278. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6405278.

[100] Alipour A, Hindle A, Stroulia E. A contextual approach towards more accurate duplicate bug report detection May 2013; :183–192URL http://dl.acm.org/citation.cfm?id=2487085.2487123.

[101] Grant S, Cordy JR, Skillicorn DB. Using heuristics to estimate an appropriate number of latent topics in source code analysis. *Science of Computer Programming* Sep 2013; **78**(9):1663–1678, doi:10.1016/j.scico.2013.03.015. URL http://www.sciencedirect.com/science/article/pii/S0167642313000762.

[102] Guzman E, Azócar D, Li Y. Sentiment analysis of commit comments in GitHub: an empirical study. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, ACM Press: New York, New York, USA, 2014; 352–355, doi:10.1145/2597073.2597118. URL http://dl.acm.org/citation.cfm?id=2597073.2597118.

[103] Brooks F. *The Mythical Man-Month, Anniversary Edition: Essays On Software Engineering*. Pearson Education, 1995. URL https://books.google.at/books?id=Yq35BY5Fk3gC.

[104] Meyer B, Arnout K. Componentization: the visitor example. *Computer* 2006; (7):23–30.

[105] Burkhard WA, Keller RM. Some approaches to best-match file searching. *Communications of the ACM* Apr 1973; **16**(4):230–236, doi:10.1145/362003.362025. URL http://dl.acm.org/citation.cfm?id=362003.362025.

[106] Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium* Aug 2009; URL http://portal.acm.org/citation.cfm?id=1595696.1595713.

[107] Graves T, Karr A, Marron J, Siy H. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on* Jul 2000; **26**(7):653 – 661, doi:10.1109/32.859533. URL http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=859533&isnumber=18656&punumber=32&k2dockey=859533@ieeejrns.

[108] Mockus A, Fielding R, Herbsleb J. Two case studies of open source software development: Apache and mozilla. *Transactions on Software Engineering and Methodology (TOSEM* Jul 2002; **11**(3). URL http://portal.acm.org/citation.cfm?id=567793.567795.

[109] Hindle, German, Holt. What do large commits tell us? a taxonomical study of large commits. *Proceedings of the International Working Conference on Mining Software Repositories* 2008; :99–108.

[110] Landis JR, Koch GG. The measurement of observer agreement for categorial data. *Biometrics* 1977; **33**:159–174.

[111] Emam KE. Benchmarking kappa for software process assessment reliability studies. *Empirical Software Engineering* 1999; **4**:113 – 133.

[112] Mockus A, Weiss D, Zhang P. Understanding and predicting effort in software projects. *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* May 2003; URL http://portal.acm.org/citation.cfm?id=776816.776850.

[113] Nagappan N, Ball T, Murphy B. Using historical in-process and product metrics for early estimation of software failures. *Proceedings of the 17th International Symposium on ...* Jan 2006; URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.5773&rep=rep1&type=pdf.

[114] Weissgerber P, Diehl S. Identifying refactorings from source-code changes. *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on* Sep 2006; :231 – 240doi:10.1109/ASE.2006.41. URL http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4019578&isnumber=4019544&punumber=4019543&k2dockey=4019578@ieeecnfs.

[115] Gegick M, Rotella P, Williams L. Predicting attack-prone components. *Software Testing Verification and Validation, 2009. ICST '09. International Conference on* Apr 2009; :181 – 190doi:10.1109/ICST.2009.36. URL http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4815350&isnumber=4815322&punumber=4815321&k2dockey=4815350@ieeecnfs.

[116] Oram A, Viega J. *Beautiful Security: Leading Security Experts Explain How They Think*. " O'Reilly Media, Inc.", 2009.

[117] Schröter A, Zimmermann T, Premraj R, Zeller A. If your bug database could talk. *Proceedings of the 5th international symposium on empirical software engineering*, vol. 2, 2006; 18–20.

[118] Anbalagan P, Vouk M. On mining data across software repositories. *2009 6th IEEE International Working Conference on Mining Software Repositories* May 2009; :171–174doi:10.1109/MSR.2009.5069498. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5069498`.

[119] Neuhaus S, Zimmermann T, Holler C, Zeller A. Predicting vulnerable software components. *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07* 2007; :529doi:10.1145/1315245.1315311. URL `http://portal.acm.org/citation.cfm?doid=1315245.1315311`.

[120] Kim S, Zimmermann T, Pan K, Jr Whitehead E. Automatic Identification of Bug-Introducing Changes. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)* 2006; :81–90doi:10.1109/ASE.2006.23. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4019564`.

[121] German D. Mining CVS repositories, the softChange experience. *"International Workshop on Mining Software Repositories (MSR 2004)" W17S Workshop - 26th International Conference on Software Engineering* 2004; **2004**:17–21, doi:10.1049/ic:20040469. URL `http://link.aip.org/link/IEESEM/v2004/i917/p17/s1&Agg=doi`.

[122] Purushothaman R, Perry DE, Society IC. Toward Understanding the Rhetoric of Small Source Code Changes 2005; **31**(6):511–526.

[123] Zimmermann T, Kim S, Zeller A, Whitehead EJ. Mining version archives for co-changed lines. *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, ACM Press: New York, New York, USA, 2006; 72, doi:10.1145/1137983.1138001. URL `http://dl.acm.org/citation.cfm?id=1137983.1138001`.

[124] von Goethe J. *Faust. Eine Tragödie*. 1828.

[125] Zimmermann T. Mining Workspace Updates in CVS. *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)* May 2007; :11–11doi:10.1109/MSR.2007.22. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4228648`.

[126] Herzig K, Just S, Zeller A. It's not a bug, it's a feature: how misclassification impacts bug prediction May 2013; :392–401URL http://dl.acm.org/citation.cfm?id=2486788.2486840.

[127] Tao Y, Sunghun K. Partitioning Composite Code Changes to Facilitate Code Review. *Proceedings of the 12th Working Conference on Mining Software Repositories - MSR 2015*, ACM Press: New York, New York, USA, 2015.

ANGABEN ZUR PERSON

Andreas Mauczka

Mollardgasse 22, 1060 Vienna, Austria

+43 699 1 971 42 92

andreas@mauczka.net

andimau

M | 31/03/1982 | Österreich

05/11/2012 – 31/05/2016

Technischer Projektleiter

Research Industrial Systems IT-Engineering (RISE)

Concorde Business Park F, 2320 Schwechat, Österreich

- Leitende Position im Bereich Qualitätssicherung und Business Analyse
- Product Owner in großen IT-Projekten
- Requirements Engineering und Test in kritischen Projekten
- Experte für Migrationsprojekte

IT, öffentlicher und privater Sektor

01/09/2008 – 05/11/2012

Wissenschaftlicher Angestellter und Software Engineer

Research Industrial Systems IT-Engineering (RISE)

Concorde Business Park F, 2320 Schwechat, Österreich

- Consultant im Bereich Software Entwicklungsprozesse
- Test Experte im Bereich Public Key Infrastructure
- Testmanager

IT, öffentlicher und privater Sektor

01/03/2008 – 31/07/2008
15/06/2010 – 14/06/2014

Projekt Assistent

Technische Universität Wien

Karlsplatz 13, 1040 Wien, Österreich

- Vortragender für Software Design und Anforderungsanalyse
- Assistent für Software Engineering und Projekt Management Lehrveranstaltungen
- Leiter Forschungsgruppe im Bereich Subsurface Engineering zur Umsetzung einer Web-basierten Anwendung im Bereich B2B

Ausbildung

01/07/2004 – 31/08/2004

Praktikum

Dr. Erich Hackhofer GmbH

Schrankgasse 16, 1070 Wien, Österreich

- Softwareentwicklung in C+, C++ und Java

IT, privater Sektor

SCHUL- UND BERUFSBILDUNG

06/10/2005 – 28/04/2008

Master of Science

Level 7

Technische Universität Wien

Karlsplatz 13, 1040 Wien, Österreich

- Projekt- und Qualitätsmanagement
- Internet Security und Kryptologie
- User Interface Design und Informationsvisualisierung

18/09/2001 – 30/06/2005

Bachelor of Science

Level 6

Technische Universität Wien

Karlsplatz 13, 1040 Wien, Österreich

- Web Engineering, Software Engineering
- Mathematik, Statistik und Analysis