Evidence-based Software Process Recovery

by

Abram Hindle

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2010

Abstract

Developing a large software system involves many complicated, varied, and interdependent tasks, and these tasks are typically implemented using a combination of defined processes, semi-automated tools, and ad hoc practices. Stakeholders in the development process — including software developers, managers, and customers — often want to be able to track the actual practices being employed within a project. For example, a customer may wish to be sure that the process is ISO 9000 compliant, a manager may wish to track the amount of testing that has been done in the current iteration, and a developer may wish to determine who has recently been working on a subsystem that has had several major bugs appear in it.

However, extracting the software development processes from an existing project is expensive if one must rely upon manual inspection of artifacts and interviews of developers and their managers. Previously, researchers have suggested the live observation and instrumentation of a project to allow for more measurement, but this is costly, invasive, and also requires a live running project.

In this work, we propose an approach that we call *software process recovery* that is based on after-the-fact analysis of various kinds of software development artifacts. We use a variety of supervised and unsupervised techniques from machine learning, topic analysis, natural language processing, and statistics on software repositories such as version control systems, bug trackers, and mailing list archives. We show how we can combine all of these methods to recover process signals that we map back to software development processes such as the Unified Process. The Unified Process has been visualized using a time-line view that shows effort per parallel discipline occurring across time. This visualization is called the Unified Process diagram. We use this diagram as inspiration to produce Recovered Unified Process Views (RUPV) that are a concrete version of this theoretical Unified Process diagram. We then validate these methods using case studies of multiple open source software systems.

Acknowledgements

I would like to recognize my coauthors on much of this work:

- Michael W. Godfrey (PhD Supervisor)
- Richard C. Holt (PhD Supervisor)
- Daniel German (Masters Supervisor)
- Neil Ernst (Fellow PhD Student and collaborator)

I would also like to acknowledge those who awarded me with scholarships:

- NSERC, as I was funded by a NSERC PGS-D Scholarship
- David Cheriton and the David Cheriton School of Computer Science for the David Cheriton scholarship I received.

I would like to thank all the people who helped motivate this work:

- Lixin Luo
- Ron and Faye Hindle
- Michael W. Godfrey and Richard C. Holt

Dedication

This is dedicated to my loving wife Lixin and to principles of Free Software as prescribed by the Free Software Foundation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software process recovery is the extraction and recovery of software development processes from the software development artifacts that developers create and leave behind. These artifacts are found in software repositories such as source control, mailing lists, user documentation, developer document and bug trackers. Some of these artifacts include: changes to source code, source code, bugs, bug reports, bug discussions, mailing list discussions, patches, user documentation, design documentation, requirements, developer documentation. These artifacts act as evidence that software process recovery analyzes in order to infer to the behaviours, the purposes, the goals, and the underlying software development process of a project.

In this thesis, we describe the area of software process recovery by describing methods and tools one can use to recover software development processes from existing software development projects. We combine many of these techniques and use them to extract software development process related events and artifacts. These events are aggregated in order to create Unified Process diagram like visualizations (sometimes called the RUP hump diagram [63]), such as our *Recovered Unified Process Views* which is described in detail in 8. In essence we use a variety of techniques to analyze the repositories of existing software systems and then present views of the underlying processes and behaviours that occur within these systems.

As developers work on a software project their efforts, their behaviours, produce evidence. This evidence usually is in the form of software artifacts such as incremental changes to source code and documentation. Often each change is recorded and annotated with a change message. Sometimes communications about the development or current use of a software project is recorded, developers and users often take part in these discussions on mailing lists and in bug reports within bug trackers. As developers create software, a digital trail of “bread crumbs” is left behind. We expect that if we can recover these trails, made by developers, that we can determine the changes they have made, infer the purposes behind such changes, and perhaps recover some aspects of the software development process these developers are following.

This extraction and recovery of software development serves multiple purposes:

- To recover and understand development processes that occurred within a software project.
- To reconcile the prescribed software development processes with the observed software development processes.
- To elicit process related information and developer behaviour without interviewing developers or relying on their perceptions and judgement.

These purposes are not unique to software process recovery. Other fields such as mining software repositories, process mining, and process discovery are concerned with similar issues, as we will explain in the following section.

1.1 Relationship to Mining Software Repositories

The field of *mining software repositories* (MSR) [88] is dedicated to exploiting and understanding software repositories and their artifacts of development, as well as inferring the relationships between them. *Process recovery* is a sub-field of MSR, that combines MSR research with *process mining* [156], the mining business processes via instrumentation, and *process discovery* [29, 31, 30], the mining software development processes via instrumentation.

We propose to improve the state of the art of process recovery by extending it with current mining software repositories research. The purpose of process recovery is to be able to retroactively recover process from project histories without having to rely on a process that has been heavily instrumented as suggested by Cook and Wolf [29, 31, 30]. Process recovery is the elicitation of underlying behaviours, informal processes, and formal processes that developers follow while building and maintaining a software project, based on evidence recovered from the artifacts that these developers leave behind. This thesis attempts to describe and execute process recovery from information left behind by developers and not rely on modifying the actual development at run-time.

Process recovery relies on the underlying artifacts to describe the development processes that have taken place. These processes and behaviours are documented within the project and recovered from artifacts such as source code, documentation, build and configuration

Process Mining	Mining of business processes at run-time
Process Discovery	Mining of software processes at run-time
Process Recovery	Mining of software processes after-the-fact

Table 1.1: Definitions used throughout the rest of the thesis

management scripts, test scripts, mailing lists, bug trackers, and revisions to all of these artifacts. The behaviours discovered and the underlying processes recovered from these behaviours can be used to characterize the processes used to develop a software project. Some of the methods used for process recovery are already used to mine business processes.

Process mining in the context of business processes, processes that define business tasks, has been heavily investigated by Van der Aalst et al. [156]. Process mining is the run-time investigation of and recovery of business processes. With respect to software engineering, and mining software for processes, Cook et al. [29, 30, 29] attempted to discover underlying processes from software projects, referred to as *process discovery*, by recording developer actions at particular times. Cook took an existing project being developed, and inserted measurement devices (instrumentation) into the process in order to record process-related information, as the project was being developed. In contrast, process recovery does not require tooling an existing process, it involves mining artifacts left behind for evidence of underlying processes. In this particular case, we respect that some projects have documented their processes, but we hope that we can help validate that these processes are being executed by recovering evidence of their use from the artifacts left behind. Thus process recovery seeks to expose and uncover processes after-the-fact, or in an *ex post facto* fashion. Table 1.1 provides a quick reference to the difference between process mining, process discovery and process recovery. In the next section we will motivate this after-the-fact process analysis by suggesting possible applications.

1.2 Application of Software Process Recovery

The usefulness of process recovery lies mainly in reporting and validating existing knowledge about a particular project. Process recovery could be used to analyze past behaviours and processes within a software project, and determine what methods of development took place. One could produce reports of the observable and recorded behaviours that occurred within an iteration and show how close these behaviours follow existing development processes and project prescribed development processes. Process recovery would give some semblance of what actions developers were taking as well was what work was done in the past. Process recovery could also aid in extracting and describing the processes and behaviours that occurred within successful projects. These processes might be useful for planning new projects. Alternatively the unsuccessful processes could be mined; perhaps

New Developers	Discover a project's process without bothering co-workers.
Fixer Developers	Integrate with projects that they are loosely associated with.
Managers	Validate the software processes being followed by their team.
ISO 9000 certifiers	Recover process in order to document the process they have.
Acquisitions	Determine what went into building a project.

Table 1.2: Stakeholders who could benefit from Software Process Recovery

there is a correlation between unsuccessful projects and the processes they use. Thus an application of process recovery can be the elicitation and validation of what processes are being followed, and what the underlying observable process is.

When we refer to the *observability* of a process we refer to the evidence that is provided and what inferences we can make about evidence. A behaviour would not be observable and its related processes would have poor observability if there was no record or evidence of such a behaviour. Behaviours such as face to face meetings or telephone conversations are often poorly recorded, thus associated processes to these behaviours might have poor observability. If a process is observable, interested stakeholders stand a chance of recovering it.

Users of process recovery would be those stakeholders who are interested in how the development of the project was executed. Potential users of process recovery are developers, managers, consultants, and those responsible for acquisitions. Developers could use process recovery to review how their coworkers were developing their current software project. Managers could verify what processes their developer's were following. Consultants could determine the effort and quality of processes used within a project, they could then determine how much work it took to implement the project. Process recovery is useful for *process verification*. Managers could investigate the development artifacts of outsourced components and try to validate if the outsourced team was following the processes they had agreed to follow. Process recovery can provide stakeholders with evidence based suggestions about what the underlying and observable processes in a project are. In the next section we will discuss various kinds of stakeholders.

1.2.1 Stakeholders

While software process recovery might not be immediately useful to each and every developer it is potentially useful for a wide variety of stakeholders ranging from managers, to new developers, to those responsible for technology acquisition. In this section we will flesh out some of the stakeholders described in Table 1.2.

Managers can benefit from software process recovery as they can use it to extract, recover and then verify if the process they prescribed is being followed by the programmers that they manage. Managers might not be as intimately involved with a software project as their developers are and might enjoy the introspection that software process recovery could provide to them, especially if it was automated. Managers care if their processes are being followed, they also care about the topics of development as well as the emphasis of development. Managers might want to understand how much work is maintenance and how much is new implementation.

Process documenters — those employees who are woefully assigned the task of documenting current processes whether it is for ISO 9000 certification, where processes are defined and documented, or the Capability Maturity Model ranking where a team is ranked on their level of process maturity. In either case the process documenter must go forth and determine the processes being followed. If they have to take part in many interviews, this will increase the costs of process documentation. Much of this work can be automated, augmented or assisted by software process recovery as software process recovery can help elicit the observable and recorded software development processes being followed.

Green developers — those who are new to a project, might want to determine what the actual software development processes and practices of a particular project are. They might want to get up to speed on the processes and methods employed by their coworkers. While they can interview their coworkers this becomes more of an issue in a globally distributed team. Perhaps software process recovery can help show new developers how development is done on the projects they have been assigned to.

Fixer developers or star programmers are often assigned to different projects and given relatively short time-spans to adjust and produce results. Often fixer developers have to jump between many projects that their valuable time is divided amongst. Often fixers become the new developer and need to orient themselves within a project. Potentially software process recovery can help them as they can quickly analyze a project and determine what the underlying culture within a project is and what observable processes do the other developers follow. Often fixers lack the luxury of being able to interview other developers and coworkers.

Investors, acquisitions consultants, and out-sourcers might be concerned about the quality of development and adherence to the software development process of a particular project. When investing in or purchasing a company with software assets it might be prudent for a consultant on behalf of the investor to investigate how the software was built as well as who built it. With respect to out-sourcing, one might wish to verify that the software development process that was prescribed in the contract was actually being followed. Perhaps process validation, via software process recovery, could be used to ensure that contractual obligations were met.

Thus all of these stakeholders can utilize software process recovery for their own purposes. The commonality among stakeholders is that they are not intimately familiar with the concrete implementation of the software project. These stakeholders are not necessarily the developers but the next section shows why developers are central to understanding the software development process of the project.

1.3 Conceptual View of Software Process Recovery

Software process recovery is the recovery of software development processes by interpreting evidence and inferring behaviours, purposes, goals, tasks and finally software development processes based on this evidence. In contrast other methods such as process mining and process recovery rely on the developers and managers to help us infer what processes they are following.

Figure 1.1 describes how software processes are composed and how developers interact with these processes. As each part of a development process has a purpose, each part is executed by a developer. When a developer implements or executes a process step relative to a goal or purpose, that developer performs a behaviour. Often behaviours produce evidence, whether the behaviour creates development artifacts or causes records to be made. We rely on this evidence to infer or suggest the related and possible behaviours, goals and software development processes. This means that without the help of developers we are trying to leverage the evidence that leave behind. This also implies that the majority of concepts in this diagram are only suggested and inferred from the evidence itself. Any behaviour that does not produce evidence is not observed, but maybe inferred by other evidence.

What this thesis is about is inferring the behaviours, purposes, and software development processes that developers followed or performed in order to develop the software development project being analyzed.

Our evidence of behaviour, purpose and process is extracted from software repositories such as source control systems, mailing list archives and bug trackers. We extract facts from this evidence, the artifacts of software development left behind. These facts are used to reason about the behaviours performed, the purposes behind the behaviours and the processes followed.

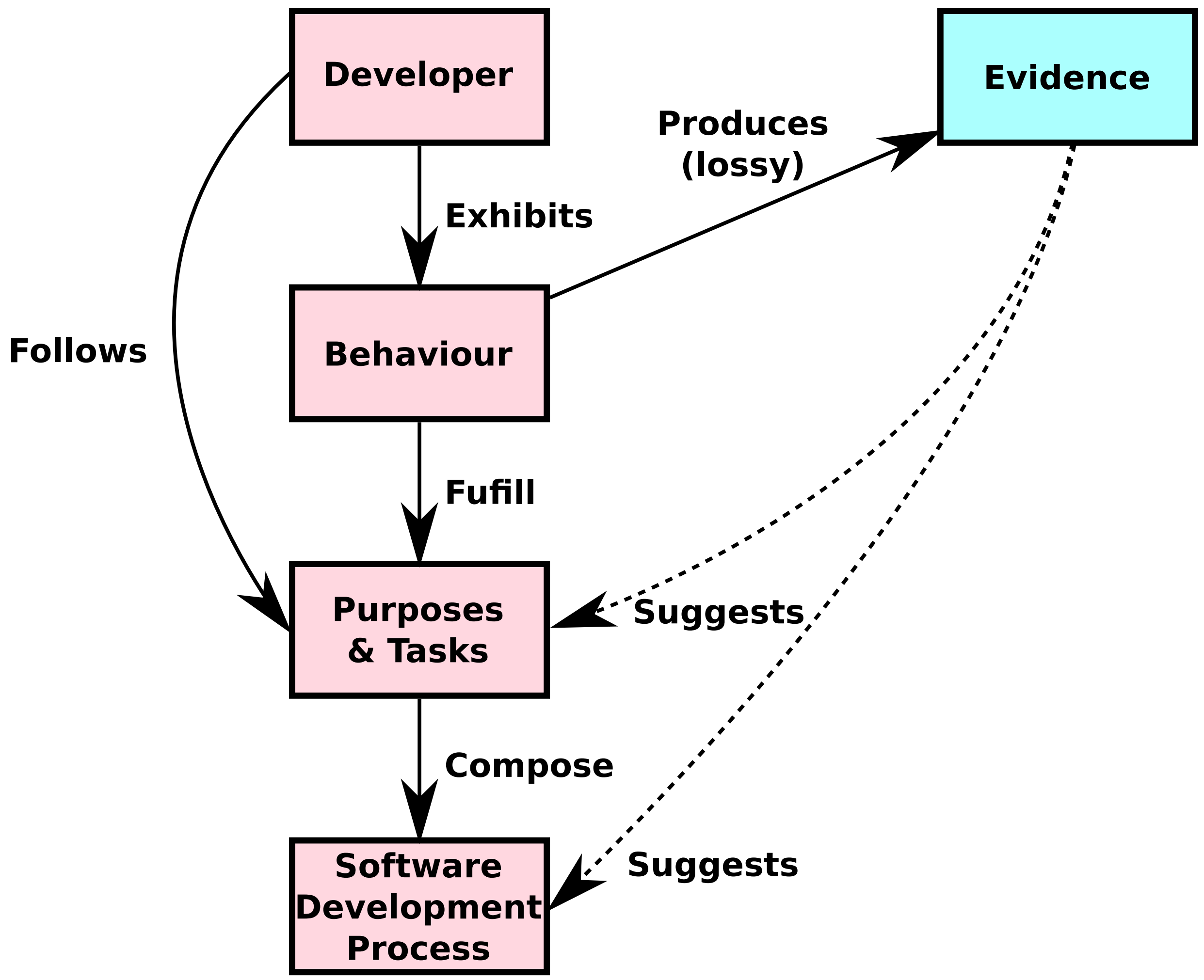


Figure 1.1: The relationship between developers and software development processes and the evidence that we rely on to recover software development processes. Developers exhibit behaviour as they attempt to complete tasks for a variety of purposes. These behaviours, tasks and purposes form the underlying software development process. Evidence is a side effect of the developer's behaviour. Using this evidence we attempt to infer the purpose and processes of software development. Thus software process recovery attempts to recover the actual development by inferring behaviour, purpose, and process from evidence.

Evidence, behaviours and purposes can be related to disciplines, activities, workflows and stages. In this thesis *disciplines*, *workflows* and *stages* refer to roughly the same concept as they are different kinds of behaviours used to build software. Some example disciplines include: requirements, design, maintenance, implementation, testing and many others. Since our case studies in the following chapters demonstrate that these disciplines are not staged sequentially but are mixed in different proportions, we feel that *discipline* is more appropriate than *stage*, as *stage* implies a singular focus.

In terms of *evidence* and *behaviour*, our Chapter 4 is about release time behaviour and demonstrates that the evidence left behind is sufficient to infer release time behaviours and processes. While Chapter 5 and 6 describe maintenance categories of changes, they also demonstrate that the evidence provides enough clues to recover the purpose or goals of a *source control system* (SCS) change based on the language in the commit message. Chapter 7, which is about extracting developer topics, leverages this evidence to demonstrate the shifting focus in a project. While Chapter 8 is about the Recovered Unified Process Views that integrate the results of the previous chapters in order to summarize the software development processes being followed.

With respect to the *purpose*, goal or tasks that artifacts are related to, Chapter 5 is about maintenance categories and demonstrates how the purposes behind changes are often observable based on the evidence provided. Chapter 6 takes the manual effort of determining purpose in Chapter 5 and automates it. Chapter 7 discusses developer topics and explores purpose in the sense of related developer topics by recovering common development threads through topic analysis of commit messages. Some of these topics are related to non functional requirements such as portability or reliability. These different methods of determining purpose are useful when trying to associate events with disciplines and workflows in software development, as described in Chapter 8.

Process related work includes Chapter 4, which studies the release time processes of multiple software projects and demonstrates that there is internal consistency within a project. This consistency is indicative of a software development process. Chapter 8 summarizes many of the other chapters and integrates their efforts into a process overview that looks like a concrete or recovered version of the Unified Process diagram shown in Figure 2.1.

From the perspective of *software development artifacts*, Chapter 4, 5, 6, 7, and 8 focus on SCS changes. Chapter 7 and 8 focus on mailing list and bug tracker related events.

Chapters that address *process observability* include Chapter 4 on release patterns, and Chapter 8 on recovered unified process views. These chapters show that there are software development processes that are recoverable based on evidence. We found that there was concurrent effort across disciplines, as demonstrated in case studies from Chapter 4 and Chapter 8. Chapter 7 demonstrates that topics of development often recur across the entire lifetime of a project, indicating that there is some repeating behaviour within software development projects.

The *disciplines of software development* are addressed across all of the chapters to differing degrees. Requirements and design disciplines are dealt with in Chapters 7 and 8 as both chapters address recovering requirements-related and non-functional requirements-related events. Chapter 4 and Chapter 5 address identifying implementation related changes whether by file type or commit log message. Testing is addressed in Chapter 4 and 8. Deployment and project management are addressed with our overview visualizations and summaries described within Chapter 8.

Thus the structure of this thesis is that if we have evidence, we can observe it and infer behaviour and processes from it (Chapter 4). Then we elaborate on this and try to annotate events by their purpose or developer topic (Chapters 5, 6, 7). Then in Chapter 8 we integrate the chapters previous to provide a general overview of the underlying software development process that we can observe based on the evidence recovered from the software repositories that we analyzed. For a more comprehensive discussion of software process recovery see Chapter 3.

1.4 Summary

In this chapter, we have proposed software process recovery and demonstrated the need for it. We related software process recovery to process mining and mining software repositories research fields. We have shown that software process recovery is valuable to a wide variety of stakeholders. Then we described how this thesis relates to the recovery of evidence from software repositories in order to infer behaviours, purposes and processes.

Our contributions in this thesis include

- A proposal for a field of research called software process recovery.
- A demonstration that we can extract some software development processes of existing software systems without instrumentation.
- The proposal, evaluation and validation of many methods of analyzing revisions, revision messages, mailing list messages and bug reports.
- An extraction of a Unified Process-like diagram called the Recovered Unified Process Views.

Our goal will be to integrate this research in order to recover underlying software processes that are observable based on the evidence recovered from the artifacts that developers leave behind. Chapter 3 describes this integration in more detail.

Next in Chapter 2 we will provide a survey of the groundwork for software process recovery. Afterwards we propose a line of research within process recovery which hinges primarily on the integration of already existing techniques and the refinement of these techniques into a relatively holistic framework for process recovery.

Chapter 2

Related Research

Our research will focus on methods, tools and techniques to aide software process recovery that are either automatic and unsupervised, semi-automatic and supervised, or manual. There is much previous relevant work related to this goal of software process recovery. This work builds upon four main areas: *process*, the formal and adhoc steps behind development; *data analysis*, the tools that we can rely on when we extract data from artifacts; *mining software repositories*, the field and the associated methods and tools that are focused on extracting information from repositories and artifacts created during development; *software process recovery*, how previous authors have attempted software process recovery and software process recovery related tasks. First off we must address what process is.

2.1 Stochastic Processes, Business Processes and Software Development Processes

Development processes, or simply processes, guide the development of software, whether they be informal and adhoc, or formal and regimented. Process is a very general word which has multiple meanings depending on the field. While our process focus is on software development processes, in this thesis we will use the term process to refer to three related concepts: software development processes, business processes, and stochastic processes. We describe each kind of process in the following subsections. Software process recovery attempts to model these processes based on the evidence left behind by developers.

2.1.1 Stochastic Processes

Stochastic processes [3] produce values or measurements that exhibit some random behaviour. These are processes that have probabilistic variation, and whose sequences of behaviour can be modelled probabilistically. Stochastic processes are much lower level than software development processes, but often represent the observed statistical properties of a data-set, such as time between changes. Stochastic processes can be described by parametrized probability distributions, such as the Poisson distribution or the Pareto distribution [65], since they involve randomness and error. The processes could be simple relationships between inputs such as time and effort and measured values like growth. With respect to software development these processes are sometimes used to model the growth of a software project.

Growth models of software, and some models of software evolution are stochastic processes as well. Lehman [98, 100, 99] described laws of software evolution relating to life cycle processes. In general Lehman's laws suggest that growth cannot persist in the long-term without control because complexity caused by growth will increase and slow down development. This was confirmed by Turksi et al. [155], who found that on some large systems growth was sub-linear. Counter examples arose in subsequent studies by Tu et al. [51] who showed that Linux in particular had super-linear growth and did not follow the laws of evolution as laid out by Lehman. Herraiz et al. [65] mined many software project and modelled growth with various software metrics and found that the distribution of the metrics used to measure growth usually followed Pareto distributions. This knowledge of underlying distributions allows us to expect certain behaviours when we measure and include these behaviours into our models.

Stochastic processes are useful as they help model underlying data, allowing us to predict or reason more effectively about what will be observed. Some processes can be modelled by well known distributions, which we discuss in Section 2.2. Stochastic processes are relevant to software process recovery because they are used to model the underlying behaviours based on evidence mined from software repositories. Stochastic processes are usually too low-level, thus they need to be abstracted and aggregated into business processes and software development processes to be useful to software process recovery.

2.1.2 Business Processes

Business processes are sequences of related tasks, activities, and methods combined, but often within the context of a business operation. These processes can be executed by actors, such as employees or software, with or without the help of software. Generally business processes describe a kind of collaboration of tasks that occur between actors. Van der Aalst et al. [156] describes *business process mining* as the extraction of business processes at

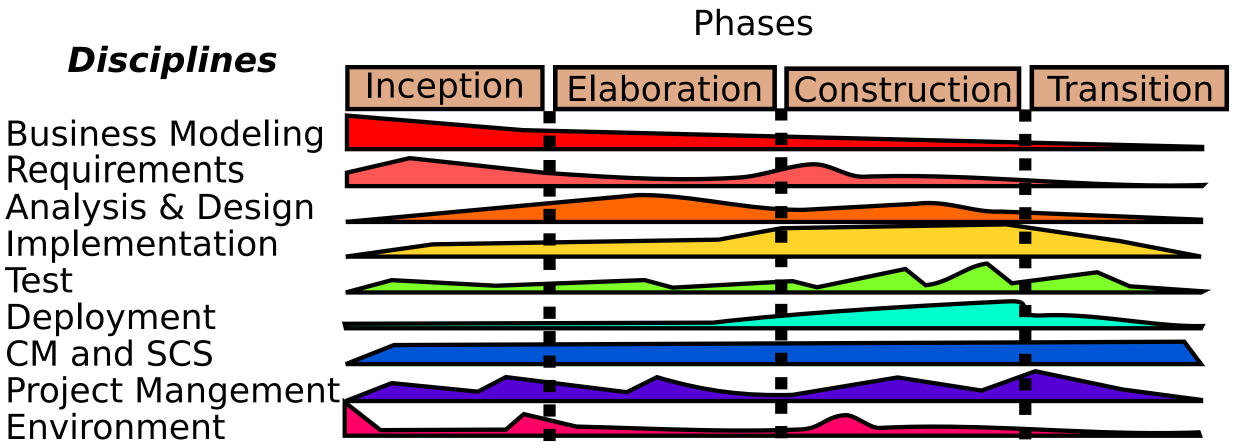


Figure 2.1: Unified Process diagram: this is often used to explain the division of labour within the Unified Process [86]. This diagram is also referred to as the UP "Hump diagram" [63].

run-time from actual business activities. Business processes can be fine-grained. Example business processes are: handling customer returns at a retail store, authenticating clients over the phone, and creating a client record. Van der Aalst deferred to Cook and Wolf when it came to process mining software projects. Software development is a kind of information work, like research, and thus is not easy to model so formally and so fine grained. While process is still important to software, we feel that the business processes described by frameworks like *BPEL* [157] (business process elicitation language) do not model software development well, as software development is often viewed as information-work. Business processes are related to software process recovery because tools such as sequence mining [89] can produce processes that are quite similar.

2.1.3 Software Development Processes

Software engineering makes use of software development processes. Software development processes are processes dedicated to analyzing a problem and designing a software solution to that problem. These are processes modelled after industrial work processes, such as the assembly line. Some researchers and practitioners hoped that software development processes would enable teams to produce software in a repeatable manner, akin to how many other engineered products were produced and designed. Software development processes are often software development methodologies. The tasks in a software development process can include just about everything: requirement elicitation, design, customer interaction, implementation, testing, configuration management, bug tracking, etc. Some processes deal with a specific part of software development such as maintenance [97]. Software development processes are seen as a way to control the evolution of a software project in order to ensure that requirements are met and the project is successful.

Software development processes are a method of control to enable the production of quality software in a reliable manner. In this thesis our definition of software development processes is:

> *Software development processes* are the composition of structured behaviours, and their purposes and intents, used to build a software system.

For example, posing software process as the behaviour of developers with respect to testing near a milestone would match this definition. Another example would be the proportionality of the efforts spent on each development-discipline over time would be a software development process. This differs from the *software development life-cycle* [144] as our focus is on behaviour and the SDLC's focus is broader.

The SDLC describes how software is often built, maintained, supported and managed. Most software development processes relate to some if not all of the various aspects of the SDLC. Classic models of software development processes include the Waterfall model [138] and the spiral model [20]. More recent software development processes include the Unified Process [86], Extreme Programming (XP) [10], SCRUM [140], and many methodologies related to Agile development [66]. See Figure 2.1 for a diagram of the Unified Process workflows over time. Most of the recent development processes focus on smaller iterations, such that design and requirements can be updated as they become more clear with each iteration. Software development processes and life cycles seek to manage the creation and maintenance of software.

Meta-processes, such as the Capability Maturity Model (CMM) [126, 1], attempt to model the processes used to create software, much like ISO standard 9000 [147] attempts to model and document how processes are executed, tracked, modelled and documented. The CMM tries to model, track, and rank software process adherence. Software process recovery could aide ISO 9000 certification and CMM adherence.

Software development models described in the literature are typically high level, they do not have many tiny fine-grained sub-tasks, but companies often have much more detailed home-brew processes. There are some processes which can be modelled in a fine grained manner though, for instance researchers have modelled how bug reports are created and handled with in a bug tracker. Their end process was essentially a state diagram of bug states [135]. Software development processes are what software process recovery tries to recover from the evidence left behind by developers. Thus software development processes model software development at a relatively high level when contrasted with finer-grained processes such as business processes.

2.1.4 Process Summary

We have covered three kinds of processes each increasing in their level of granularity and generality. At the low level we have stochastic processes that seek to describe the processes behind the measurements and the data itself. At the intermediate level we have business processes which describe fine grained actions and tasks taken to fulfil a goal. Then at the high level we have software development processes which group together different efforts into a general process. We seek to extract all three of these kinds of processes from existing artifacts in an effort to recover the underlying processes that were executed to build the software project.

2.2 Data Analysis

In order to automate process recovery we will need to extract and process the data recovered from the software development environment. This data comes in many forms, from many repositories, as described by the mining software repositories community [88]. The main tools we use and describe include: statistical analysis, time-series analysis, machine learning, sequence mining, natural language processing, and social network analysis.

2.2.1 Statistics

Statistics [3] help us describe data and model data, and they are an integral part of data mining. In the rest of this section, we will summarize some commonly used statistical methods; the reader may choose to skip over this. The main statistical tools we use are distributions and summary statistics. Distributions are characterizations of how values, such as measurements, are spread out across their possible range. Distributions describe the relative frequency of values occurring or having occurred within a data-set. Some distributions have been modelled as functions or shapes. These distributions are parametrized, so we can generate them from a function and a set of parameters. Some common distributions include: the normal distribution (bell curve), exponential distribution, Poisson distribution, Pareto distribution, power-law distribution, Zipf distributions, etc. Distributions can be summarized or characterized by summary statistics. Summary statistics are metrics that describe aspects of a distribution such as the average, the median, the variance, skew, kurtosis, etc. These summary statistics can be used to describe the shape and expected range of values of a distribution without the need to show the distribution itself. Quartiles are another useful measure, quartiles are a set of four ranges of elements that share an equal number of elements, quartiles help define the range and spread of a distribution. The value between the middle two quartiles is about the median of the data-set.

Distributions and summary statistics tell us about the underlying data and help us model the data. In order to better model the data it is useful to see how similar distributions or models are to each other.

The statistical tools we use for comparing distributions include: the t-test, the χ^2 test, Kolmogorov Smirnov test, the Lilifors test, cosine distance and Euclidean distance. These distribution similarity tests can be used to indicate if underlying distributions are similar or if a distribution matches a well known distribution like a exponential distribution, a Poisson distribution, a power law distribution, etc. The Lilifors and Kolmogorov Smirnov tests are especially useful for dealing with software-related data because they are non-parametric, which means that they can be used to compare data distributions that are not Gaussian (normal); this can be contrasted with the t-test and chi^2 tests, which can be sensitive to other kinds of distributions. Lilifors and Kolmogorov Smirnov tests are based on measuring the largest distance between two cumulative distribution functions (the integral of the probabilistic distribution function). These tests allow us to compare distributions to each other, but sometimes we want to search for distributions that are similar in order to compare models to the underlying data.

One common way to compare models is to use linear regression. Linear regression tries to model the data by finding the best fitting line through our data; this is accomplished by trying to minimize the R^2 values, which indicate how much unexplained variance there is in our data compared to our best matching linear model [79]. These statistical methods allow us to compare and reason about data-sets, create model of the data-sets, and validate these models against the data-sets.

2.2.2 Time-series analysis

A time-series is a set of data points plotted over time. MSR research often deals with time-series, and much of the data collected from software repositories is data that occurs across time, which means the data has temporal components. The underlying data could be an event, it could be an aggregate of events, it could be a measurement at a certain time. The analysis of this data is called *time-series analysis*. Hindle et al. [69] discussed spectral analysis applied to time-series. Spectral analysis, using Fourier transforms, allows for underlying periodicities to be recovered from signals such as changes per day. What this means is that if a behaviour recurs regularly, such as weekly, that it might be detectable with spectral analysis. This spectral analysis is similar to the *auto-correlation* used in time-series analysis [64]. Auto-correlation attempts to look for underlying repeating behaviour by comparing a signal to time-shifted versions of itself. Much of the analyzed data from repositories is time-series data, utilizing these tools allows us to deal with time and look for potentially repeating patterns across time.

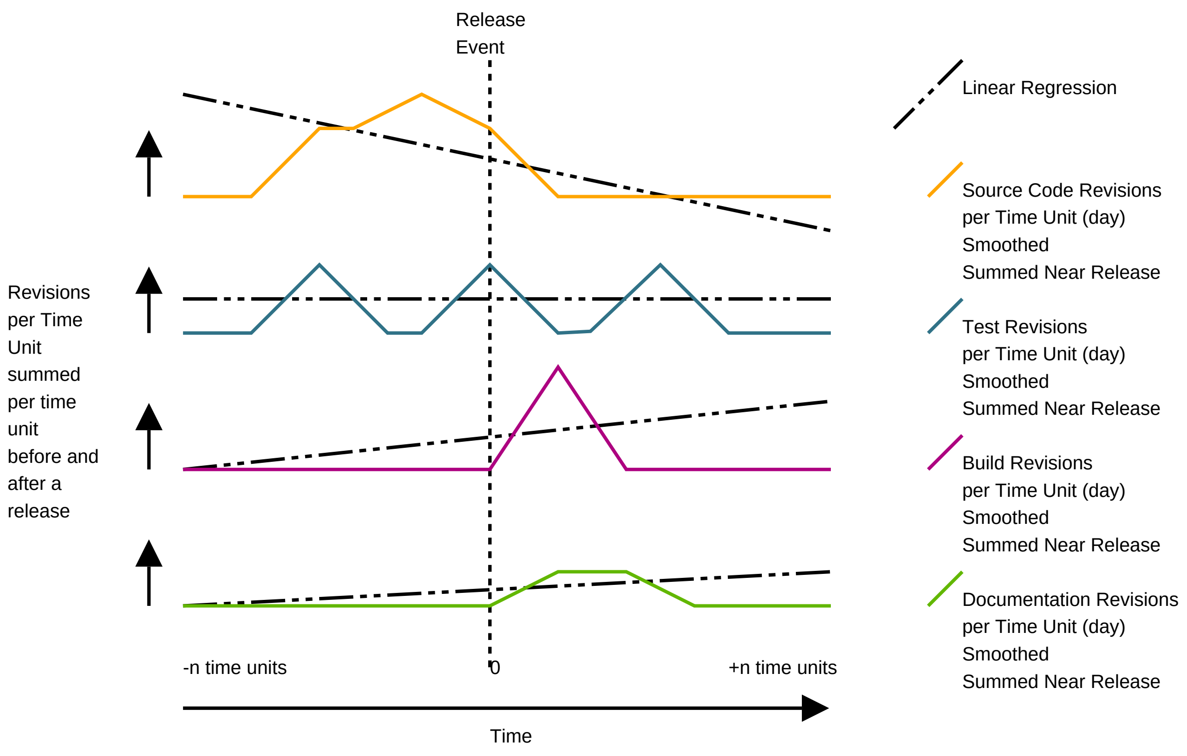


Figure 2.2: A project's change history split into four streams: source changes, test code changes, build system changes, documentation changes.

Time-series analysis also relates to *longitudinal studies* [141]. These are studies of data and measurements over time. Longitudinal studies can employ multilevel modelling, which is a mixture of statistics and time-series analysis. Multilevel modelling first requires that we model measurements of entities (e.g., changes to a file, changes made by an author) over time. These are our first-level models. Then we can aggregate the parameters of these models and produce a second-level model which generally describes all of the underlying entities and the variances of each parameter. A common scenario for longitudinal data analysis is to model the performance of children in a classroom. The first-level models are those of the children and their performance throughout the school year. The second level model is the general model of the children altogether, the average rate of change of performance and the variance within the children's models. Longitudinal studies and multilevel modelling allow us to reason about general events like the major releases of a project, but also about the instances such as a single major release of a project.

We have used a rudimentary form of multilevel modelling [75] to describe how individual releases behave compared to the general trend of release behaviour within a software project. Figure 2.2 gives an example of how we analyzed behaviour around release time. Thus we have already utilized time-series data, Fourier analysis, and multilevel modelling in our own research [75, 69]. All three of these methods are valuable for process recovery because they allow us to describe patterns that occur over time, find periodic behaviours, and model individuals and groups of entities. This allows us to recover stochastic processes and identify processes by mining repeating behaviours.

2.2.3 Machine Learning and Sequence Mining

Machine learning [84] provides a powerful set of tools that helps process recovery by enabling automatic classification of entities based on historical facts. Machine learning is attractive because it can often utilize training data (the past) to classify the new data. It also enables bootstrapping, where one invests some work classifying changes manually and then delegate the rest of the work to the machine learner. Machine learning is a broad topic and we have used many machine learners to analyze and classify the maintenance purposes of large revisions [73]. See Figure 2.3 for an example of distribution of large changes in multiple open-source projects.

Sequence mining [89] attempts to find repeating sequences in streams of discrete entities. Sequence mining is valuable in process recovery because it can be used to find common chains of actions or events that occur within a project. Sequence mining can be used to mine processes at different levels of aggregation from fine grained events to the common sequences of phases of an iteration. Related to sequence mining is item-set partitioning [142], where similar behaviours are grouped along time in order to partition a time-line by observed self-similar behaviours. Thus machine learning, sequence mining

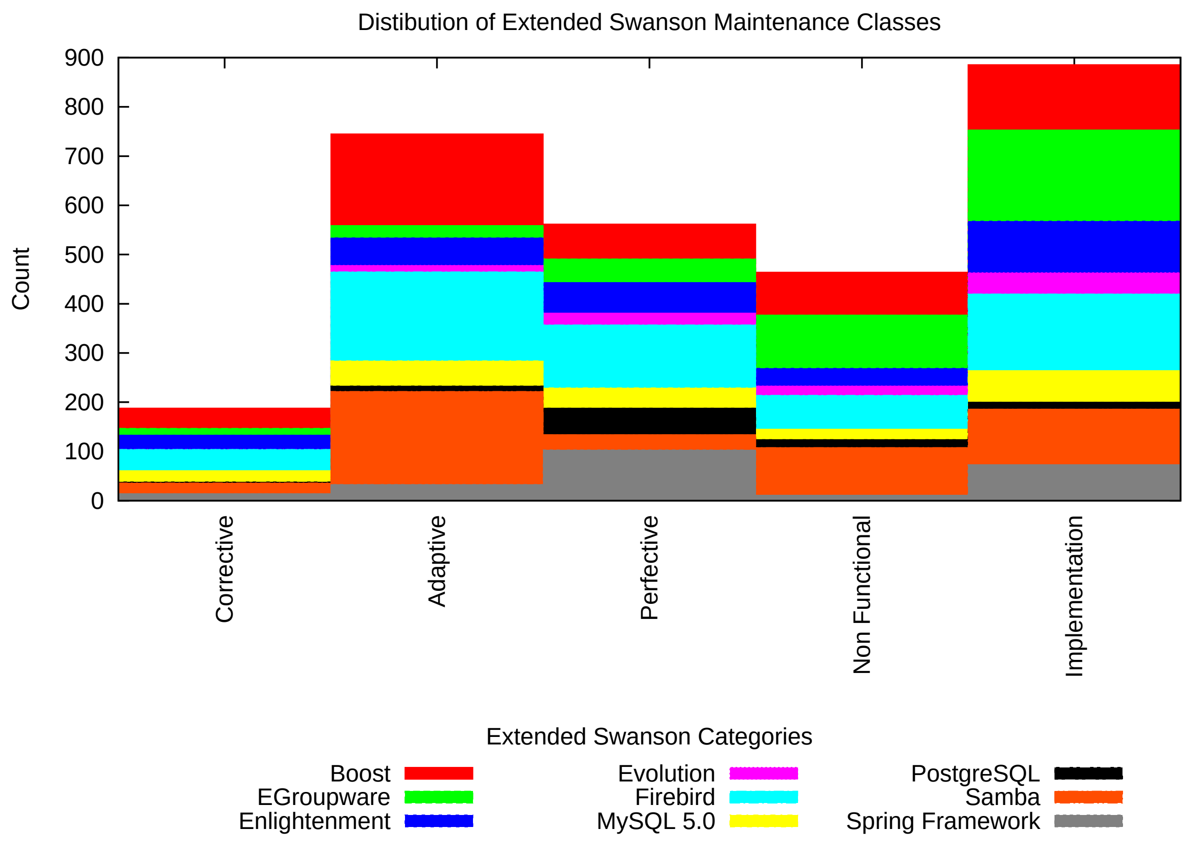


Figure 2.3: Proportion of large commits per project per Swanson Maintenance Category

and item-set partitioning help us infer new facts that not yet expressed. Machine learning can be used to help identify phases, while sequence mining can be used to find business processes inside of stochastic processes, and item-set partitioning can help identify when discontinuities occur thus perhaps helping us discover the end of a phase or an iteration.

In this thesis we utilize machine learning to attribute certain classes of behaviours to certain artifacts, or to relate concepts by classification to artifacts such as bug reports and source code changes. Often machine learning relies on natural language processing to produce learner friendly training-sets from textual data.

2.2.4 Natural Language Processing

We often have to analyze text and source code. Text is generally unstructured. One method to analyze text and even source code is to leverage techniques used in the field of Natural Language Processing (NLP) [90]. The most common NLP technique to analyze text is to abstract it as a word distribution. A word distribution is a word count, with optional stop word removal. Word counting produces a word distribution per each entity. These word distributions allow various other tools to process the natural language text, whether they act as input to a machine learner, or are used to represent entities in topic analysis.

Natural language processing (NLP) [90, 145, 67] deals with processing language in the form of speech or text into a computer-based representation of that speech or text. This processing might be used to better understand the underlying message, or it can be used to discriminate between low-level natural language entities like words in a sentence or discriminate between high level entities like documents and sets of documents. For the purposes of this research we utilize NLP tools and methods in order to relate and process text messages left behind by developers. We also use similar structures and tools to process source code as well.

The basic NLP tools that we use are word distributions, stemming, and stop word filtering. Word distributions are a method of abstracting a block of text into counts of words, which can be further normalized by the size of the message. Word distributions are useful because one can apply a distance metric such as Euclidean distance, cosine distance, or the difference in cumulative distribution function (CDF) (derived from the Kolgoromov Smirnov test) in order to compare two messages. Stemming transforms words into their root forms. For instance a gerund like *biking* could be reduced to *bike*, and *dragged* to *drag*. Stemming cleans up tenses and modifiers to words, in an effort to make word distributions smaller and more similar if they cover similar concepts. Stop word filtering is another technique which removes words that are not important to the current analysis. For instance stop words, that could be removed when we are mining for concepts

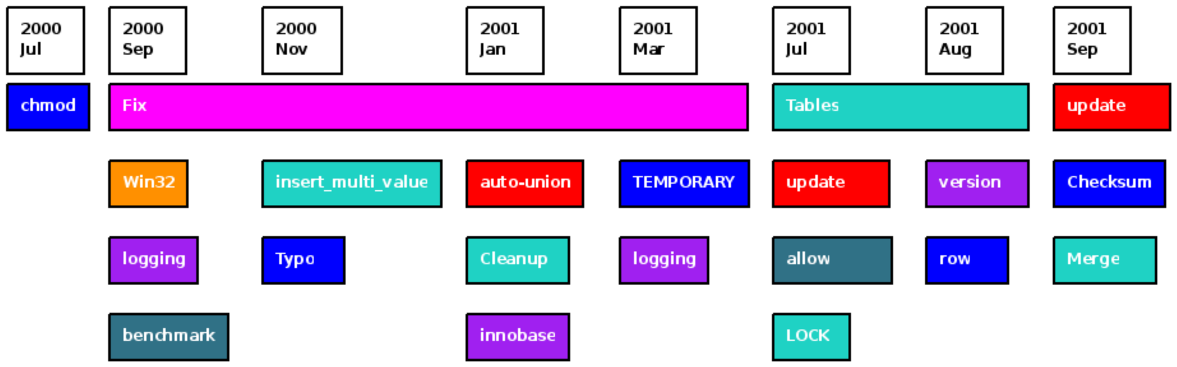


Figure 2.4: A manual example of topic analysis applied to MySQL 3.23 revisions. [80]

or modules, could be definite articles like *the*, other modifiers, or language keywords like *else*. Word distributions also serve as inputs for other NLP related techniques like topic analysis using LDA [17], LSI [109, 128], or ICA [34, 52].

Topic analysis or concept analysis is the automatic discovery and mining of topics, usually modelled as word distributions, that pervade a set of messages. For instance, a newspaper is often organized into sections by overall topic, and if a topic analysis algorithm were applied to all articles within a newspaper, we might expect that topic analysis techniques might suggest topics that matched the main sections of a newspaper, such as life, entertainment, international, local, national, etc. Latent Dirichlet Allocation (LDA) [17] is an unsupervised topic analysis tool that is popular within the mining software repositories community [146, 103, 104, 102]. Other similar topic analysis techniques include latent semantic indexing (LSI) [109, 129], independent component analysis (ICA) [52], principle component analysis (PCA) [34], probabilistic Latent Semantic Indexing (pLSI) [83], semantic clustering [95, 96], etc. Sometimes source code and bug reports often serve as input to topic analysis tools, where as our own work used LDA to mine topics from commit comments [80] (an example of extracted topics appears in Figures 2.4 and 2.5). Figure 7.2 describes how one could use a tool like LDA or LSI to analyze commits.

NLP techniques are useful to process recovery because they recover information and associations from unstructured text data or data that is being treated as unstructured. These associations allow us to associate artifacts with different kinds of topics and potentially work-flows such as requirements, design, testing, or bug fixing.

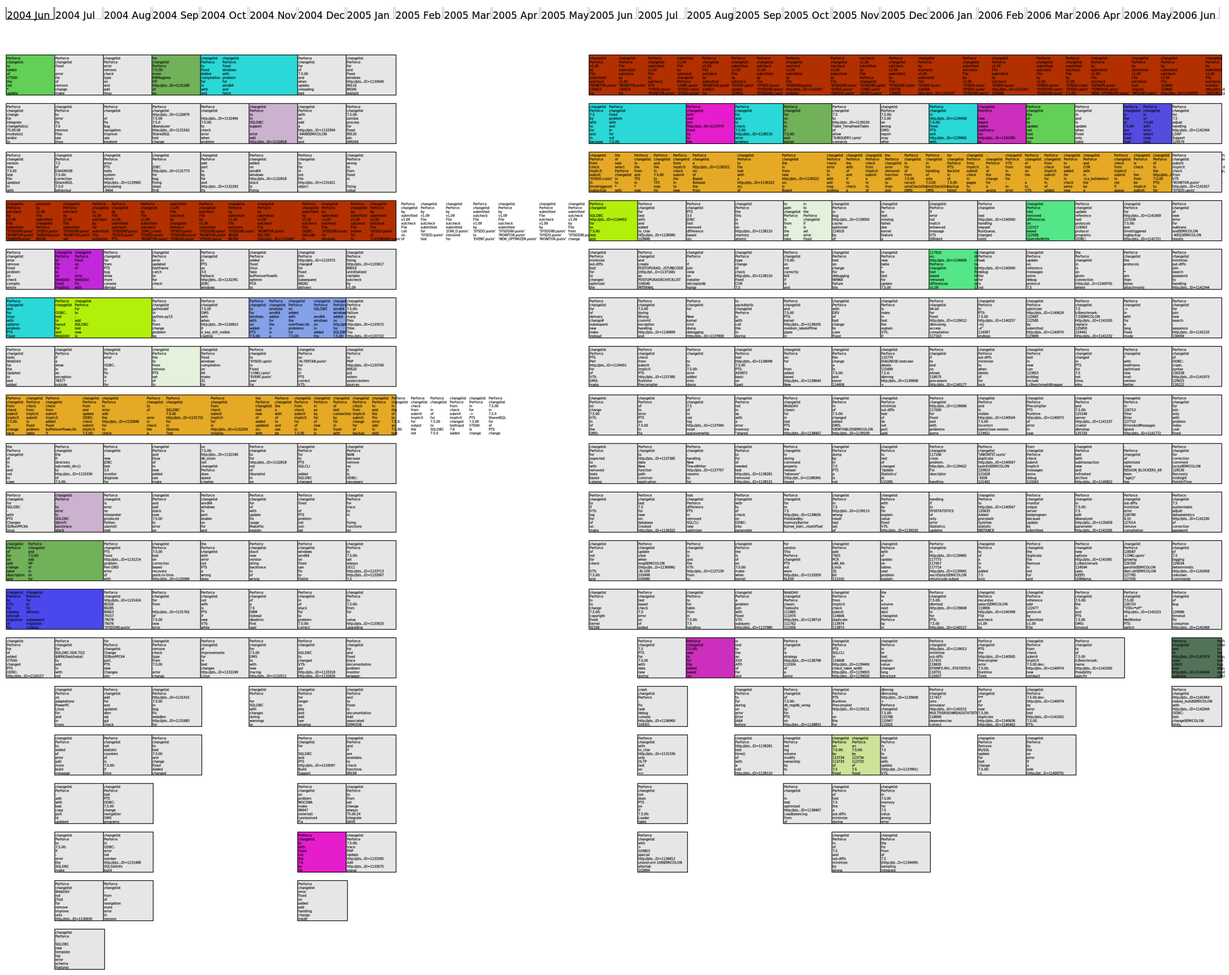


Figure 2.5: Automatic topic analysis applied to MaxDB 7.500. This compact-trend-view shows topics per month for MaxDB 7.500. The x-axis is time in months, the y-axis is used to stack topics occurring at the same time. Trends that are continuous are plotted as continuous blocks. Trends with more than one topic are coloured the same unique colour, while topics that do not recur are coloured grey. The top 10 words in the topics are joined and embedded in box representing the topics. No stop words were removed. [80] See Section 7.1.5.

2.2.5 Social Network Analysis

Social network analysis [158] is the structural analysis of social interactions between people or actors. A social network models the interactions between people and entities as a graph with entities as nodes and their relationships as arcs. These graphs and their interactions between nodes are measured using a wide variety of graph metrics such as centrality, connectedness, whether or not a node bridges two clusters, etc. The mining of social networks from software repositories is a popular field of research, whether it be social networks extracted from emails [13, 16] or other source code related artifacts [159, 15]. Social network analysis allows to study the structure of interaction, thus it can aid in process recovery by elucidating the relationships between developers, which might help us associate developers with project roles.

2.2.6 Data Analysis Summary

In this section we provided an overview of the tools that can be used to recover processes from artifacts left behind by developers. Statistics aides in modelling underlying data so we can describe the data and reason about it. Time-series analysis lets us reason about data that has temporal qualities. Machine learning and sequence mining allows software tools to make decisions and extract new facts based on the past. Natural language processing helps us deal with the large amount of unstructured and semi-structured natural language artifacts (such as messages or bug reports) that developers and users leave behind. Social network analysis lets us see the social interactions between actors and entities, allowing us to discover developer roles. All of these data analysis techniques have been leveraged in the field of mining software repositories and thus are quite relevant to process recovery.

2.3 Mining Software Repositories

Software process recovery relies on software related artifacts left behind by developers. These artifacts often appear in software repositories. Mining software repositories [88, 92] refers to the investigation and mining of data within software repositories such as source control, mailing lists, bug trackers, wikis, etc. Kagdi et al. [88] provide a survey and taxonomy of approaches and studies that exist within the mining software repositories (MSR) community [60]. Work in this area seeks to answer questions such as: which changes induce bugs or fixes [143], what information do software repositories tell us, how could new repositories be more useful, and what are the underlying properties of software, software evolution, and changes to software. Software process recovery is a sub-field of

MSR because it relies on the repositories and the mining techniques that MSR literature employs in order to reason about underlying processes.

Much of the software and the software repositories that are mined in MSR literature is *Free/Libre Open Source Software* (FLOSS), this is due to the availability of FLOSS. Since FLOSS development artifacts are typically freely available, we can have repeatable studies on the same corpus, thus allowing researchers to validate their results against the results of others, and to compare different approaches easily. Some researchers have studied and measured the general characteristics of Open Source Projects [23]. Mockus et al. [121] described the underlying methodology of FLOSS projects such as Apache and Mozilla. Much of the information they used was gleaned by mining the source control repositories of these projects. Thus much MSR research is executed on FLOSS.

2.3.1 Fact extraction

When one approaches a software repository often one has to reason about the data inside of it. An important first step is to extract this data from the repository. This step is called fact extraction. Many researchers have covered various details of how to extract facts from a SCS and store them into a database to be queried later [105, 49, 46, 41, 160, 61, 162]. Some issues that fact extracting handles: data cleansing, tracking identities, grouping revisions into commits, resolving symbols, etc. Fact extraction allows us to export a fact-base from a repository so we can reason about it in other contexts. Fact extraction is necessary for process recovery because one needs to build up a fact-base in order to reason about underlying processes.

2.3.2 Prediction

We can use these extracted facts to help predict the future. Some developers and managers seek to predict what will happen next. Much work has been done on prediction by Girba et al. [50], where they attempt to predict effort much like how weather is predicted, based on past information. Others such as Hassan [62] attempt to model change propagation in software systems via measures such as co-change. Herraiz et al. [65] attempted to correlated various metrics with lines of code (LOC) and growth. These metrics provided a power-law-like distribution which could be could used to predict metric values. One potential use of MSR research is to use past data to predict the future in order to aid project planning.

Much work has gone into bug prediction, whether predicting buggy changes [94], buggy files, or bug-prone functions, there are entire conferences essentially dedicated to bug prediction [139]. Much research in these conferences utilizes MSR related data to reason about the locations of future bugs. Many of these predictions rely on software metrics.

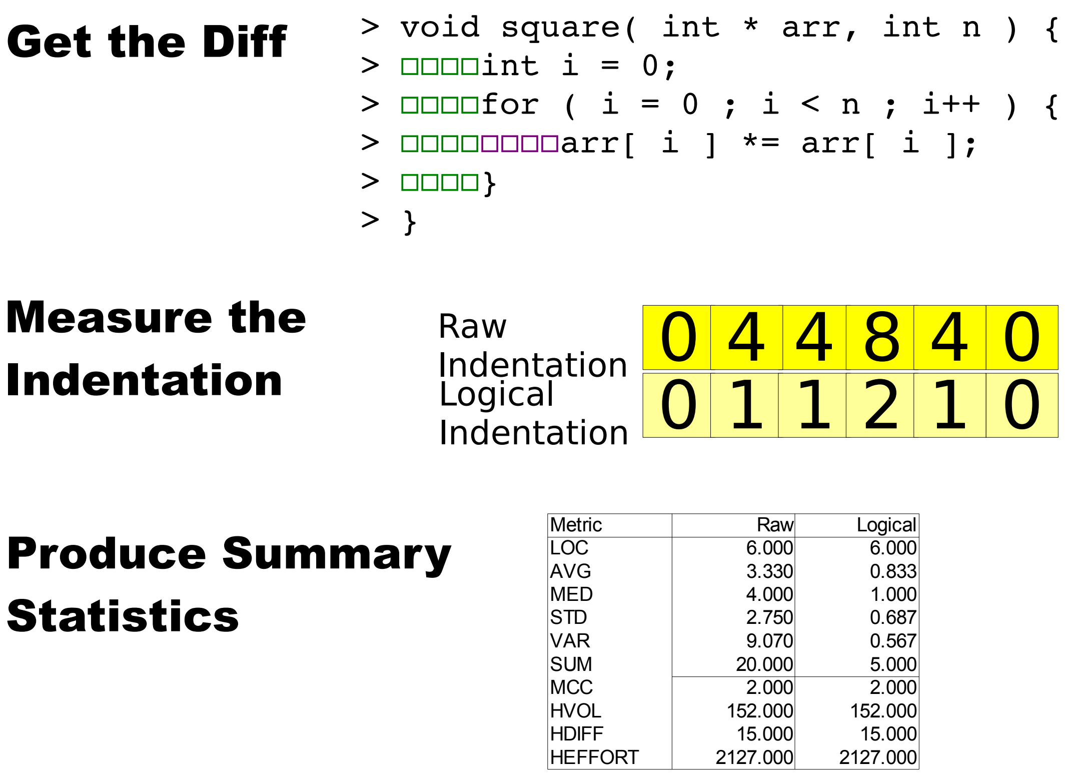
2.3.3 Metrics

In order to describe observations quantitatively we need to count and measure them, software metrics are meant to allow us to count and measure properties of software systems and their artifacts. Some metrics are related to process and deal specifically with measuring how many requirements a team has fulfilled so far [59]. Some metrics do not relate to the code itself, but about relationships that are observed external to the code.

Software Metrics — classic software metrics [39] include lines of code (LOC) [137, 65], and complexity measures such as Halstead's complexity [57] and McCabe's cyclomatic complexity [111]. There are object oriented (OO) metrics [130, 11, 58] like fan-in and fan-out, measurements of class hierarchies, calls-in and calls-out. Some have tried to correlate various software metrics with maintainability [27, 125] and have produced the maintainability index that attempts to indicate the maintainability of software. These classic software metrics describe underlying code but often for MSR related work they need to be extended by time.

Evolution Metrics — there are MSR and evolution related metrics [100, 115, 116, 114] that often deal with changes or project versions. There are also evolution metrics that focus on the deltas rather than the differences between versions. Measurement of changes, whether they be revisions, diffs, deltas, or structural deltas has been investigated in work by Ball et. al [9], Mens et al. [115], Draheim [35], German et al. [48], and Hindle et al. [77, 76]. Herraiz et al. [65] studied how metrics related to LOC over time. Hindle et al. [79] investigated the indentation of source code, particularly source code that appears in revision diffs, they found that the variance of indentation in a diff correlated with classic complexity measurements such as McCabe's complexity and Halstead's complexity. An example of how indentation metrics are measured is shown in Figure 2.6. Thus many MSR related metrics measure change itself, but there are other metrics which measure temporal relationships or couplings between entities.

Coupling Metrics — some metrics are concerned with historical and logical coupling or co-change [43, 42, 62] between entities. Co-change is where two files or entities change together, and how often they change together. Co-change is also referred to as logical coupling [43, 42]. Co-change and coupling indicate cross correlations between entities. Zaidman et al. [161] studied the co-change between entities and tests, while Hindle et al. [75, 68] studied the co-changes between changes to source code, test code, build systems and documentation at release times. Coupling is useful within process recovery because if one wants to characterize a behaviour, one can use measures of co-changes across different classes of entities at the same time.



```
> void square( int * arr, int n ) {
> □□□□int i = 0;
> □□□□for ( i = 0 ; i < n ; i++ ) {
> □□□□□□□□arr[ i ] *= arr[ i ];
> □□□□}
> }
```

Raw Indentation	0	4	4	8	4	0
Logical Indentation	0	1	1	2	1	0

Metric	Raw	Logical
LOC	6.000	6.000
AVG	3.330	0.833
MED	4.000	1.000
STD	2.750	0.687
VAR	9.070	0.567
SUM	20.000	5.000
MCC	2.000	2.000
HVOL	152.000	152.000
HDIFF	15.000	15.000
HEFFORT	2127.000	2127.000

Figure 2.6: How indentation metrics are extracted and measured.

2.3.4 Querying Repositories

Once the facts have been extracted and the metrics suites run, we are left with a lot of information. There are many things an end user might want to do with this information: they might want to see related artifacts, they might want to query for structural qualities of changes, they might want to have changes described in a high level manner. If an analysis program can be rewritten as a query it saves both the researcher and the end user time.

A query system for developers to find hints to related documents was designed by Cubranic et al. [32]. Hippikat [32] is a SCS query system which attempts to link multiple "software trails" from different sources into one search engine for developers. Hippikat is more like a search engine rather than say a relational database.

The SOUL system [22] allows for querying programs by structure and example, over time, but also allows users to pose queries in a Prolog like language, giving the query a lot of flexibility. This allows entities to be queried in a relatively unconventional manner. Related to SOUL is Semmle, SemmleCode and .QL, the semmle query language [101], together these tools provide an object-oriented query system to staticly analyze code. Semmle is used to query the underlying source code of a project by utilizing static analysis. There are also other logic-based query systems related to MSR.

Hindle et al. [72] created and defined a query language, Source Control Query Language, for querying data from SCS using first order and temporal logic queries. These queries were created to check for invariants within a project, as well as generally query a repository for change patterns. Hindle et al. built a system which uses first order and temporal logic to find entities, Others have written systems which take existing data and then use logic to describe the change.

Kim et al. [93] have studied how to described fine grained changes succinctly. They used a descriptive grammar of first-order logic and automatic inferencing in order to make a concise and small description of a change, which described whether it included certain files or if it modified certain structures within a program. Descriptions could be similar to: "All subclasses of Mapper were changed except for AbstractMapper". These concise queries work well when designing reports for end users who want high-level overviews.

Querying and inference are valuable tools to look for certain behaviours within a software repository. Inference allows for concise descriptions of changes while querying enables for concise extraction of patterns and results. Conciseness might be important to process recovery when used in the context of creating reports.

2.3.5 Statistics and Time-series analysis

Some MSR research seeks to quantitatively reflect and describe the behaviour within a software repository. One way to do this is to utilize summary statistics, software metrics, and time-series analysis. Because so many repositories have entities with temporal aspects, time-series analysis and the extraction of temporal patterns is quite useful.

Large distribution studies have been executed by Herraiz et al. [65], Capiluppi et al. [23], and Mockus et al. [119]. In these studies metrics and summary statistics were used on numerous FLOSS projects. These studies provided more of a static model of what to expect from software measurements and were not so much about evolution.

With respect to time, much work is done on time-series and MSR related data. Israel Herraiz et al. [64] worked on ARIMA models of the number of changes. ARIMA models are meant to model and predict time-series data using techniques like auto-correlation. Herraiz studied many FLOSS projects and tried to characterize the ranges of the ARIMA model's parameters that could model these projects. Related work in mining recurring or repeating behaviour was done by Antoniol et al. [7], where they attempted to mine time variant information from software repositories using linear prediction coding (LPC). LPC is often used in speech audio compression. Hindle et al. [69] described how to pose these time-series and recurrent behaviour problems in the context of Fourier analysis, that is to treat time-series as signals and apply signal analysis techniques to the data itself. Time-series analysis helps us reason about changes and behaviour over time but if we have many entities with many time-series we can use tools like multilevel modelling.

Release Pattern Discovery [75, 68] is much like multilevel modelling [141]. Each release has its own parameters for its models (a linear regression or a simple slope of changes across an event). Release pattern discovery attempted to communicate the behaviour of developers at release time by breaking up fine grained changes, recorded in SCSs, into four streams: source code changes, test code changes, build changes, documentation changes. This enables the end user to describe a behaviour around a release in terms like, "source code changes generally increased across the release, while testing changes were constant. There were usually no documentation changes before a release but many documentation changes after a release". This multilevel modelling allowed us to talk about individual releases as well as all releases. Software process recovery requires time-series analysis in order to reason about the fine-grained aspects of the various artifacts being analyzed.

2.3.6 Visualization

Visualization, with respect to mining software repositories, often attempts to reconcile the added complexity of time with already complex data. For instance when MSR and UML diagrams are combined you can potentially have a new UML diagram per each revision to the source code. Visualization seeks to deal with this complexity by leveraging visual intuition and reasoning to aid the understanding of the underlying data.

Much software visualization research is related the visualization of software design and architecture. Usually architecture is represented as graphs, where modules are nodes and relationships between modules are arcs. Sometimes hierarchical containment relationships are used to deal with tree-like data. Rigi [122], Shrimp [148] and LS-Edit [150] are interactive graph visualizers which typical display hierarchical relationships of a software's architecture using both containment and arcs. Rather than architecture, some researchers have attempted to visualize the social structures within a project [124]. Most of these tools provide a static, although interactive view. These tools usually do not support visualizing graphs that evolve over time.

One way to visually model time and evolution is by animation. Visualizations that use 3D commonly use animation for navigation. Gall et al. compared releases via 3D visualizations [44]. Marcus et al. applied 3D visualization to source code [110]. While others have used animation to show the progress of time, for instance researchers have used VRML to animate software evolution matrices [118]. Beyer et al. [12] used animation and software evolution metrics to produce storyboards of changes to files. Other kinds of non-3D visualizations typically use graphs.

Much visualization has been applied to graphs and temporal-graphs, ranging from D'Ambros's radar plots [33], which indicated recency with distance, to Lungu et al. [107] whose system displayed versions of graphs that tried to maintain the locality of the changes. Many other tools attempt to draw temporal graphs of software metrics and changing architecture [127, 132, 151]. We have our own animated graph drawing tool, YARN (Yet Another Reverse-engineering Narrative) [70, 82], which produces videos of the changing call-in and call-out dependencies between modules and shows these dependencies cumulatively over time. For an example of YARN see Figure 2.7.

Time-lines provide a static, non-animated, visual histogram of counts or amounts over time. Evograph is an MSR related project that maps time to time-lines rather than animations [40]. Time-lines are used in Figure 2.1 of the Unified process [86] to indicate effort per workflow over time. Note that this diagram of effort is not extracted, it was expected, process recovery could aide in regenerating a diagram like this to be used in a report. Heijstek et al. [63] have created concrete views of the UP diagram using a suite of IBM tools including ClearQuest. Heijstek et al. benefited from the fact that the IBM tools used mapped very closely to the Unified Process. Heijstek et al. had access to different repositories that we did not have access to. Visualization whether it be static graphs, animations or time-lines is useful for exploratory studies of the evolution and process recovery. In general if a pattern can be spotted visually it should be extractable via automated methods, thus visualization is more about exploration and understanding.

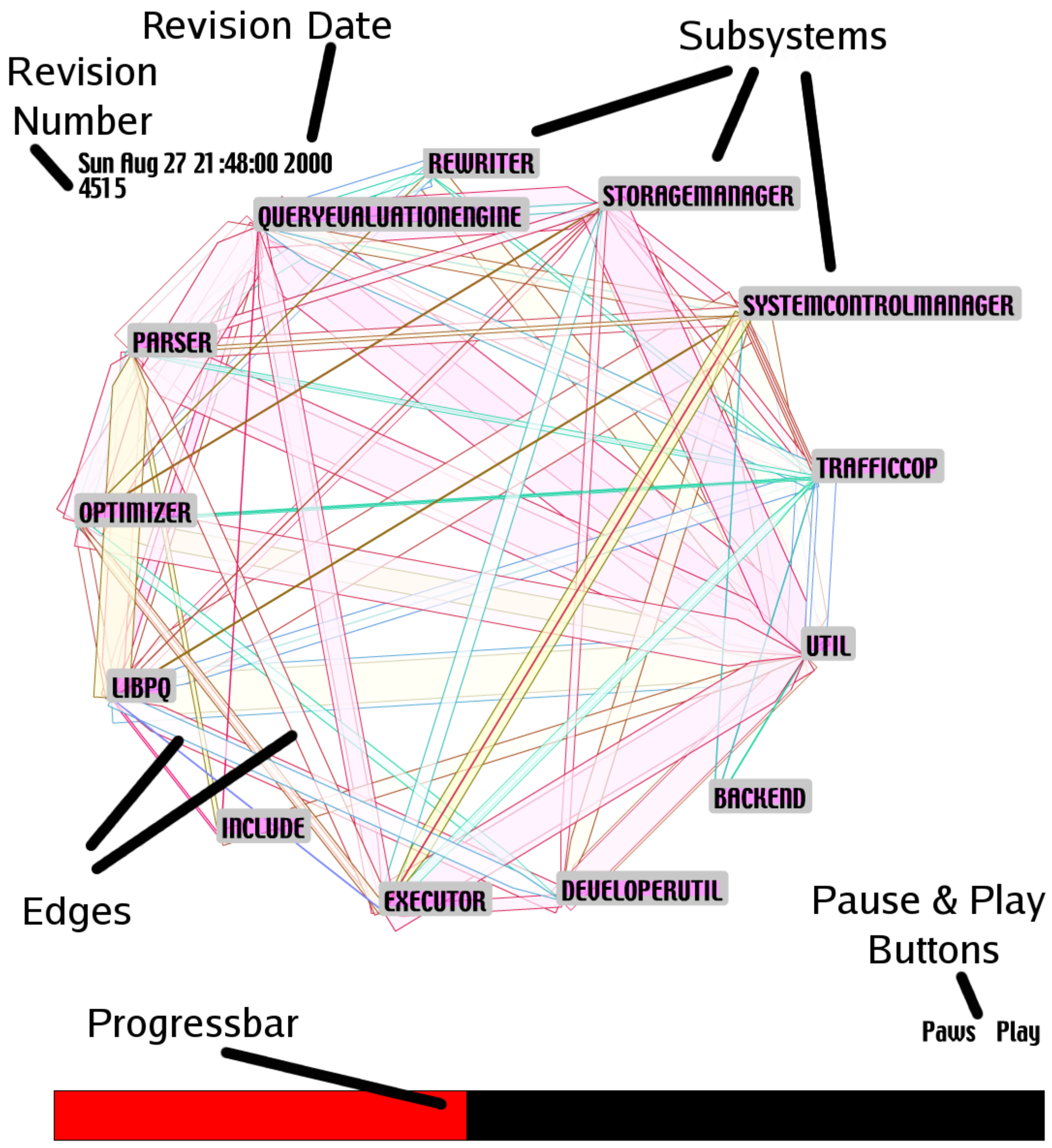


Figure 2.7: YARN Ball: PostgreSQL module coupling animation, the thickness of the arc and direction indicates how many times one module calls another. Modules are the boxes along the edge of the circle [82].

2.3.7 Social Aspects

Some studies of software repositories focus mostly on the developers and users, and the communication between developers and users. Much of the socially related MSR research analyzes mailing list repositories [134, 13, 14, 16].

Rigby et al. [134] attempted to characterize the conversational tone of developers on the Apache mailing list using psychometric textual analysis which contained word lists associated with certain emotional aspects.

Most other socially related MSR research dealt with social network analysis, the network of communications, dependencies and relations between developers, their artifacts and the users. Bird et al. [13, 14, 16, 15] create dynamic and static social network graphs and analyze them for behaviours like when a group of developers enters or a leaves a project, when a user becomes a developer, who becomes a developer, etc. Social network analysis and psychometric analysis are useful for relating stakeholders, such as developers to a time-line, their artifacts and their coworkers artifacts. The study of collaboration gives us hints to how the team works together and maybe their underlying roles and processes.

2.3.8 Concept Location and Topic Analysis

Often when mining various data-sources, one wants to associate high level concepts and ideas with fine-grained entities, *concept analysis*, *concept location* and *topic analysis* help with this task. *Concept analysis* finds concepts within documents and tries to relate them to source code. *Concept location* takes an already known high level concept such as a bug and tries to find the documents that relate to that concept. For example, when fixing a bug, how and where do the concepts in the bug report map back to the source code? *Topic analysis* attempts to find common topics, word distributions, which recur in discussions extracted from documents such as messages, change-logs, or even source code. Topics produced through the topic analysis of development artifacts can be called *developer topics*.

Research on formal concept analysis and concept location [108, 109, 128, 129] has often utilized LSI or semantic clustering [95, 96], while topic analysis has used both LDA and LSI [102, 146, 128] For instance, Lukins et al. [146], used LDA for bug localization. They would produce an initial model and then use a bug report as an example of a document they wanted to see. The example document would be broken down into a linear combination of topics and then the most similar documents would be retrieved. This is visually explained in Figure 7.2. Grant et al. [52] and have used an alternative technique, called Independent Component Analysis [34] to separate topic signals from source code. We used LDA to analyze change-log comments [80] in a windowed manner where we apply LDA to windows of changes and relate those windows via similar topics.

Concept location and topic analysis are particularly useful for finding the relationships between disciplines like bug fixing or maintenance and the artifacts that mention them. This kind of analysis applied in a MSR setting allows us to generalize about the topics and concepts being tackled within a project.

2.3.9 MSR Summary

Mining software repositories takes software engineering problems and complicates them with another dimension: time or versions. Mining software repositories is inherently linked with data analysis and process. Much MSR research attempts to model and explain behaviours and the resulting programs that come from this evolution of artifacts. More importantly MSR is generally the application of different data analysis tools in order to mine artifacts extracted from software repositories. These software repositories range from bug trackers, to source control systems, to mailing lists. Software process recovery attempts to federate many of these techniques for the purposes of modelling processes from this data.

2.4 Software Process Recovery

Software process recovery, as explained in the introduction, is a sub-field of mining software repositories dedicated to extracting processes, at different levels of granularity, from software repositories without requiring that the underlying software development be modified, instrumented, or tooled to aide extraction. Software process recovery is related to two fields outside of MSR: process mining and process discovery. Process mining attempts to discover business processes from already running formal and informal processes within an organization. Process discovery attempts to apply process mining to software by modifying the development process in order to build up metrics which can be used as evidence of the underlying process. Software process recovery takes a different approach, in order to discover process, one analyzes the existing artifacts left behind by stakeholders such as developers and users.

2.4.1 Process Mining: Business Processes

Business process analysis attempts to recover business processes from existing systems. Researchers such as Van der Aalst [156], attempt to mine workflows and business processes while the process occurs within an organization. Process mining has its own term for fine grained analysis: *delta analysis*. Much of process mining includes workflow analysis, that consists of applying or extracting finite state machines or petri nets from the observed activities. Process mining attempts to model processes using states extracted from observations.

Some within the MSR community have worked on extracting business processes from software [55, 54, 53]. We find that business processes are too fine-grained for many software processes, as business processes often model the purpose of the software, and not necessarily the development of the software itself. Although process mining is quite relevant, process recovery seeks to analyze the development of the software.

2.4.2 Process Discovery

Process discovery is an attempt to take process mining and apply it to software by modifying the existing software development process of a project to accommodate and record metrics used to discover underlying processes. In the field of process discovery Cook [28]. has described frameworks for event-based process data analysis [30]. In [31], Cook uses multiple methods to measure and correlate measures extracted from processes. Some of the metrics used were string distances between a process model and the actual process data, workflow modelling, and Petri-nets. Cook also discusses grammar inference, neural networks and Markov models as applied to process discovery [29]. The value of process discovery is that much of the groundwork for models of processes for software development have already been implemented. Process discovery is relevant because it is an instrumented method of extracting software development processes from a live project. Process recovery is similar to process discovery but with a different perspective and goal, process recovery does not seek to instrument live projects, instead it seeks to recover processes from the software artifacts of a project.

2.4.3 Process Recovery

By taking process discovery and applying it to software repositories and artifacts already left behind by developers we have process recover. Thus process recovery is a mix of MSR, process mining and process discovery methods combined and applied evidence left behind.

Within the MSR community, many researchers have tried to extract processes. For instance, Ripoche and Gasser [135] have investigated Markov models for use in FLOSS bug repair. They created state transition diagrams that represented the probabilistic state transition model of the Bugzilla bug database for the Mozilla project. Their ideas are derived from the process modelling methods of Cook [28], specifically the idea of using Markov chains to describe the state transitions. This kind of process is closely related to business processes in terms of detail.

Software process recovery at a software development process level has been discussed by Jensen and Scacchi [87]. They describe various ways of mining information from FLOSS projects to facilitate process modelling. The focus in this line of research was mining web resources to "discover workflows" rather than source control repositories. They use ontologies and directed graphs to describe work flows. Most importantly Jensen et al. uses *a priori* data, that is if they know something, they provide it to their tools and their analysis. In other research, German manually mined process documentation and mailing lists in order to describe the processes that GNOME project developers used to create and manage the GNOME desktop environment [45].

Heijstek et al. [63] analyzed multiple industrial projects that used a suite of IBM tools such as ClearCase and ClearQuest. With access to documentation repositories such as ClearCase, Heijstek et al. produced Unified Process diagrams using effort estimation. Most notably Heijstek et al. had access to time-sheet data as well. In Chapter 8 we implement a similar larger scale study except the repositories we rely upon are not so well matched to the Unified Process.

Thus there has been a range of MSR-related investigations into process recovery. The research has ranged from the automatic to the manual. Different researchers have extracted processes from bug repositories, project websites, and mailing lists. What is left to be done is a more general integration and general approach to finding iterations, to finding sub-phases, and identifying process from a project's artifacts.

2.4.4 Software Process Recovery Summary

We have reviewed three different but related branches of research: (1) the fine-grained process mining of business processes, discussed by van der Aalst et al. [156], which are often too fine grained to be useful for software; (2) process discovery, discussed by Cook et al. [28]; (3) process recovery research [135, 87, 45, 55, 54, 53, 63] from within the mining software repositories community. What distinguishes software process recovery from process discovery and process mining is that process discovery and process mining instrument existing processes in order to observe the process. Software process recovery does not rely on instrumentation or even live projects as it derives behaviour, purpose and process from the artifacts that were left behind.

2.5 Summary

In this chapter we have reviewed previous work that has inspired much of the work within this thesis. We covered four main areas: processes, analysis, MSR, and software process recovery.

We covered work into processes ranging from statistical processes, stochastic processes, to software development processes. We showed how other kinds of processes relate to software development. We reviewed the data analysis tools such as statistics and time-series analysis for events, and natural language processing for dealing with textual data.

After covering the groundwork of processes and analysis we then reviewed research from the area of mining software repositories. Our work relies on MSR research because it specifically mines repositories for behaviours related to processes. MSR covers a wide range of repository analysis tasks such as extraction, metrics, statistics, visualization, and topic analysis.

After we covered the MSR research we tied this research back to software process recovery by showing the evolution of process mining, discovery and recovery. We motivated software process recovery by demonstrating that process mining and process discovery often required the instrumentation a live system while software process recovery did not.

Chapter 3

Software Process Recovery: A Roadmap

In order to recover software development processes from evidence, such as software artifacts, we need to show that we can find the building blocks of process: behaviours, purposes and goals. Based on these behaviours and purposes we can observe or derive software development processes. Yet many behaviours are not observable because they are not recorded and do not leave evidence behind, such as phone-calls and meetings. The mixture of behaviours within a software project is the de facto software development process of a project. A *prescribed process* is a process that a developers agreed to follow, these processes are often dictated to programmers by their managers. In many cases these de facto software development processes might differ from the existing prescribed development process of a project. Thus the three main questions that we have to address are: based on evidence from software repositories can we observe the underlying behaviours? Can we observe the process from these behaviours? And can we recover the purpose or goals behind many of these events and behaviours? To answer these questions we need to understand the composition of a software development process.

Software development processes consist of multiple stages, tasks, workflows, disciplines and milestones. In order for us to crack the nut of software process recovery we will need to handle the many dimensions of software development and software development processes. These dimensions include the stages or disciplines of development, such as iterations, maintenance, and requirements. These dimensions are related to a multitude of issues facing software development.

Software development addresses many issues regarding requirements, design, implementation, testing, quality assurance, portability, non-functional requirements (NFRs), tools, development environment, project management, communication, and many more. Thus

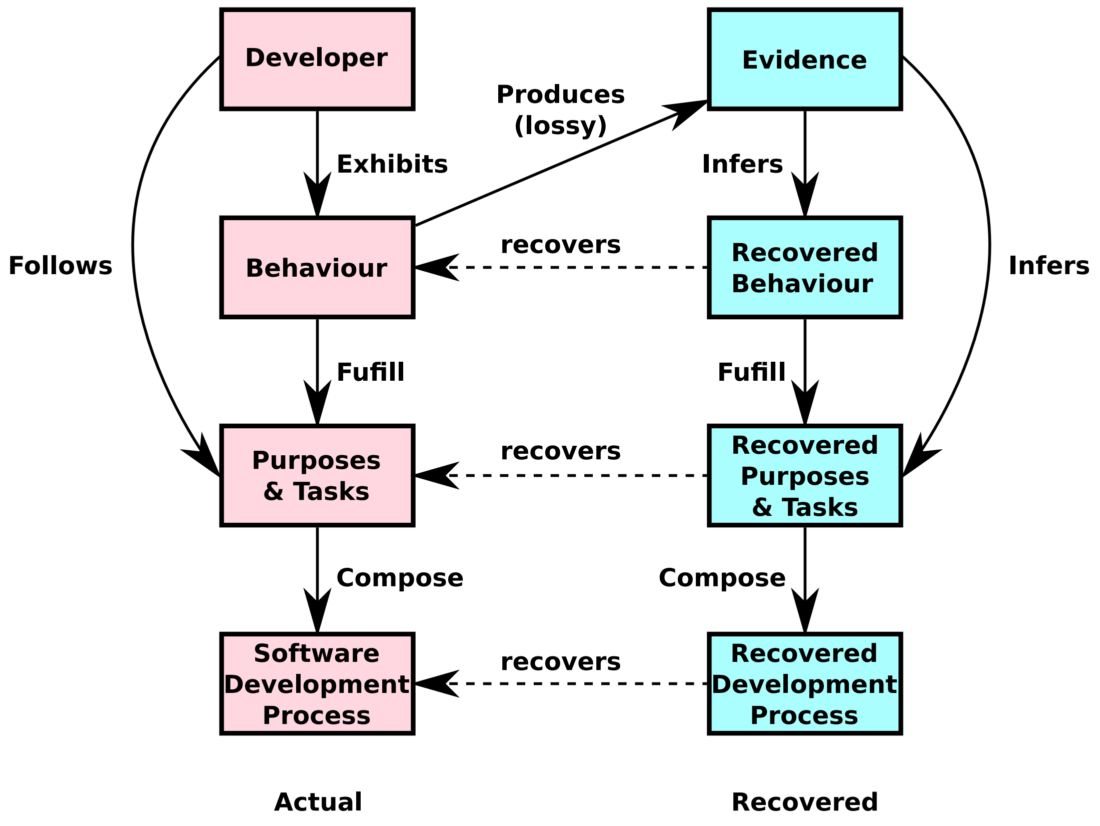


Figure 3.1: An illustration of the reality of development versus what we can infer from evidence left behind. Developers exhibit behaviour as they attempt to meet goals that compose the software development processes they follow. Evidence is a side effect of the developer's behaviour. Using this evidence we infer the behaviours that the developer followed as well as the purpose behind this behaviour. By combining both recovered purpose and behaviour we can infer the recovered process. Thus software process recovery attempts to recover the actual development by inferring behaviour, purpose, and process from evidence.

a software development process is often expected to address some of these issues. To address these issues developers often perform behaviours that are relevant to addressing these issues.

Our focus is to characterize and summarize behaviour over time. One way to do this is by breaking down events into multiple workflows or disciplines. In Chapter 8, we summarize workflows by combining the previous chapters into one coherent summary of software processes via a kind of diagram called Recovered Unified Process Views. This diagram is a concrete version of the Unified Process diagram in Figure 2.1. So how do these issues and behaviours relate?

Our model of software process recovery is described in Figure 3.1. This figure is a more concrete version of Figure 1.1 as the software process recovery fact extraction and inference is made more explicit. We have to address two worlds, the actual development of a project and the model of it we infer from available evidence. The actual development of a project produces evidence as a side effect of the behaviour of the developers. These behaviours take place in order to fulfil tasks or purposes that compose the underlying software development process. Based on evidence produced by behaviours we can infer and derive developer behaviour and the tasks or purposes behind this behaviour. This evidence is often recorded in software repositories like source control systems, mailing list archives and bug trackers. This evidence is raw and often needs to undergo the process of fact extraction and inference. Once behaviour, tasks and purposes have been inferred we can attempt to recover the software development process. The rest of this chapter will relate the following chapters with the concepts from Figure 3.1.

We propose software process recovery and within this thesis we describe many techniques that can be used to elicit various aspects of software development, and then we integrate many of these techniques into a larger overview. Each of these techniques approaches software process recovery in a different manner and for a different purpose. The three themes of software process recovery are: *behaviour*, *evidence*, *purpose*. These themes have to be addressed before any processes are recovery from repositories such as source control systems, mailing list archives and bug trackers.

We need to acquire and analyze the evidence left behind. Thus we need to determine if any behaviour or process is observable based on evidence such as source control revisions. In Chapter 4 we discuss how we can observe process by analyzing source control revisions and looking for changes to source code, test code, build code or documentation.

To determine the purpose of evidence such as a change, one must interpret the evidence. In Chapter 5 we investigate the content of commits themselves and to what tasks, goals and purposes they relate to. We show that maintenance and implementation are taking place in parallel.

Then in Chapter 6 we utilize the work in Chapter 5 in order to automate the classification of changes by their purposes. We effectively demonstrate our ability to automatically classify some changes using machine learning and natural language processing.

During software development developers often discuss particular topics that face the project. In Chapter 7 we address the topics of development that are local to a project and those that recur. These *developer topics* are topics or issues that developers discuss in commit log messages and other discussions. These topics are often specific to a project, but many are general issues that face most software systems. Topics that are project specific often relate to unique requirements and design inherent to the project itself, these topics distinguish one project from another. These project specific developer topics might be issues relating to the implementation of the project, or bugs. Topics that can relate broadly across many projects include *non functional requirements* (NFRs) such as efficiency, portability, reliability, and maintainability. Similarly *local topics*, topics that do not repeat globally during a project's history, often focus on a specific issue passionately for a short period of time, while recurring topics within a project often relate to "ilities" and non-functional requirements such as correctness. Later in Chapter 7, we further reconcile these topics and automatically label those topics that are found across many projects. Specifically we label those topics that relate to non-functional requirements (NFRs).

Once we have our evidence gathered, our behaviours and purposes inferred we can try to aggregate all of this information to suggest what are the underlying software development processes of a project. In Chapter 8 we integrate much of this work in order to produce a framework and methodology that yields us a time-line-like perspective on the observable aspects of process within a project.

Thus we will demonstrate for many FLOSS projects that:

- software development processes are observable based on evidence left behind (Chapter 4);
- we can find the behaviours within a repository by categorizing changes into various classes both manually and automatically (Chapter 4 and Chapter 5);
- we can elicit the purpose of changes (Chapter 5);
- we can extract the topics of change (Chapter 7);
- we can identify topics common to software development (Chapter 7).

In the next three sections we outline how we approach software process recovery using different perspectives. These perspectives are necessary as software development is complex. The first perspective is the software artifact perspective in Section 3.1 that looks at software process recovery from the data that is available to us. Section 3.2 takes a process-oriented perspective, as we address issues about concurrent behaviour, and the mixture of effort between disciplines. Section 3.1 takes a look at process recovery from a software development process perspective.

3.1 Software Artifact Perspective

We will execute software process recovery on artifacts extracted from three kinds of repositories: source control systems, mailing list archives, and bug trackers. Source control systems (SCSs) store artifacts such as changes to source code, change comments, revisions, commits, documentation, tests, assets, build files and other files related to a software project. The artifacts of mailing lists are messages, discussions and threads. Bug trackers contain artifacts such as bug reports, issue tickets, bugs, bug discussions, and patches. There are more repositories one could use but for our purposes and for the projects we studied these are the three main accessible archives. Within the context of FLOSS the next most common repositories would be documentation repositories such as wikis.

The next few sections will motivate our research using the software repositories that we extract our software artifacts from:

- Source control systems (Section 3.1.1 and Chapters 4, 5, 7, 8)
- Mailing list archives (Section 3.1.2 and Chapter 7 and Chapter 8)
- Bug trackers (Section 3.1.2 and Chapter 7 and Chapter 8)

3.1.1 Source Control Systems

Source control systems are version controls systems (VCS) for source code and software projects. Chapters 4 to 8 deal with version control systems in differing amounts. Most have a commit or revision focus, which is often referred to as fine grained analysis. Some chapters like 5 and 7 are rely on the descriptions that programmers give their changes. The rest of this thesis relies heavily on SCS data.

Chapter 4 demonstrates that process is observable based on evidence held within a SCS.

Chapter 5 describes what kind of commits and the purposes of these commits contained within a SCS.

Chapter 7 describes the kinds of topics that are discussed in SCS commit messages.

Chapter 8 uses the time-series data of the events within a SCS.

Most of our research relates to source control systems because the majority of the data we have is from these repositories. The artifacts contained within source control system are very focused and on topic as they deal specifically with implementation issues and details. Other repositories such as mailing list archives and bug trackers, discussed next in Section 3.1.2, can be more off-topic whether it is general discussions, discussing future changes, or proposing new features which might never be implemented. Often discussions are not found in SCS comments but on mailing lists and in bug reports.

3.1.2 Mailing list and Bug Trackers

When people communicate via a mailing list their discussions are often stored within mailing list archives so that they can be accessed later. Mailing lists are effectively a broadcast discussion mechanism. While bug trackers are somewhat similar, they are often not email based, and are more formal than mailing list discussions. Each bug report is usually named, and has a description and meta-data associated with it. Sometimes discussions, much like a mailing list discussions, are attached or included with a bug report.

Mailing list archives and bug trackers are often very similar. The FreeBSD project use the bug tracker *gnats*, which is am email based bug tracker. This is not always the case, often bug trackers are ticketing systems that allow attachments and discussions.

For our purposes of software process recovery we want to leverage the data within mailing list messages and bug reports. Most of this information is natural language text and often the immediately most useful thing to do with such information is to categorize it or classify it. In Chapter 7 we primarily analyze the source control system commit messages and then label these messages by the NFRs that they are related to by using word-bag analysis. Chapter 8 describes how we leverage this technique and apply it to bug trackers and mailing list archives. Mailing list archives, bug trackers, and source control systems can be analyzed together via traceability links.

These repositories are our primary sources of evidence. Thus given these three main types of repositories and the artifacts within, we have enough information to be able to recover some of the software development processes that a project might follow.

3.2 Process Perspective

Software development processes instruct practitioners on how to plan and produce software. Software development processes often suggest how to order actions, how often certain action should be repeated, and how to manage the complexity that is software development and project management.

Processes deal with issues such as releases, tags, behaviours at certain times, iterations, milestones, repeating behaviour and concurrent work. In our case we suspect many processes are just mixtures of efforts at different times, depending on the focus of development and the current stage in the project's life cycle.

In the next few sections we will address how this research fits into process related issues of software process recovery:

- Evidence of processes (Section 3.2.1 and Chapter 4 and Chapter 8)
- Iterations, releases and tags (Section 3.2.1 and Chapter 4)
- Behaviour at a certain time (Section 3.2.3 Chapter 4, 5, 7)
- Repetition and consistent behaviour around events (Section 3.2.2 Chapter 4)
- Concurrent effort across disciplines (Section 3.2.2 and Section 3.2.3 and Chapter 4 and Chapter 8)

3.2.1 Evidence of Software Development Processes

One of the first questions we have is: Can we observe software development processes based on evidence extracted from the repositories and the artifacts that developers leave behind? A short answer is yes, some aspects are observable: Chapter 4 demonstrates via a very simple partitioning and time-series analysis that a repeatable process is observable within many FLOSS projects. One way to investigate repeating behaviour is to study the multiple releases within a project.

Software releases are an important part of a project's software development process. Most processes talk about releases whether it is one monolithic march to a release, such as in the waterfall model, or a cyclic iterative model where releases are at least once per iteration. Since releases are important demarcation points because they are chosen by the developers as a milestone where the project is readied for consumption. Thus releases are good points around which to analyze a project for indications of process because they are intentional process events. Chapter 4 focuses on tags and releases and demonstrates there are self similar release patterns within projects. This self-similarity is important because it indicates that both iteration and repetition of processes are somewhat observable based on analysis of the SCS revisions.

3.2.2 Concurrent Effort

Concurrent effort is when effort is split across many disciplines at one time rather than focusing on one particular workflow or discipline, for instance if testing and implementation occurred at almost the same time we would classify that as concurrent effort.

Chapter 4 investigates if work is done in stages or in a concurrent mixture of efforts. The projects studied in Chapter 4 all showed concurrent efforts. Chapter 8 is concerned about modelling and describing the concurrent effort expended on a project by partitioning events and software artifacts into different time-lines.

3.2.3 Process and Behaviour

Software development processes concern the behaviour and emphasis on issues such as quality. While Chapter 4 deals with release-time behaviour, Chapter 5 shows what a particular change actually is by describing its behaviours and purposes.

Chapter 7 looks at the issues discussed within a project and relates them to software quality topics. Thus we demonstrate that there are repeatable behaviours. We also show that in some cases we can determine the NFRs related to the changes.

This ability to track concurrent behaviour and observe process is particularly useful to software process recovery because it allows us to deal with software processes that are truly mixtures of efforts rather than discrete stages. The next step is to integrate general processes with software development.

3.3 Software Development Perspective

The Software Development perspective concerns different kinds of workflows, disciplines, work and artifacts that go into developing software. Essentially all of the things that software development processes describe. In order to recover software development processes we have to recover events, artifacts and behaviour relating to software development itself.

The following sections relate software process recovery and software development:

- Requirements (Section 3.3.1 and Chapters: 7, 8)
- Design (Section 3.3.1 and Chapter 7, 8)
- Implementation (Section 3.3.2 and Chapter 5, 4)
- Testing (Section 3.3.2 and Chapter 4)

- Deployment (Section 3.3.2 and Chapter 8)
- Project management (Section 3.3.3 and Chapter 8)
- Maintenance (Section 3.3.3 and Chapter 4 to 8)
- Quality assurance (Section 3.3.3 and Chapters: 7, 8)

3.3.1 Requirements and Design

The ultimate goal is to recover requirements because they indicate and dictate what the software is about. We will handle a small subset of requirements related issues that we can observe from the evidence we have. Although the waterfall model is more of an abstraction for discussion rather than an actual concrete process to be followed, it still suggests requirements modelling should come early in the life-cycle of a project, while other models suggest that requirements can be handled iteratively often at the beginning of each iteration, but sometimes intermixed with implementation and design.

Chapter 7 investigates developer topics recovered from source control commit messages. These topics often relate to requirements being addressed in the code. The second part of Chapter 7 goes further and labels relevant topics by non-functional requirements that they are related to, such as efficiency and portability.

In Chapter 8 we demonstrate how we can track requirements that occur during development using word-bag analysis demonstrated in Chapter 7. In our case studies in Chapter 8 we found that we could extract some design discussions as well.

Given requirements and design, the next step is to relate these discussion to the actual implementation and maintenance of these requirements and designs.

3.3.2 Implementation, Testing, and Maintenance

Implementation is the stage where the software is truly created based upon the design and requirements. Implementation is the creation of concrete and executable software. Typically implementation refers to the creation of actual source code, which is the first stage of a feature's concrete existence. Chapter 4 and Chapter 5 investigate tracking implementation, testing, and maintenance.

In Chapter 4 to track implementation we partition revisions in source control by source code changes, and non-source code changes. In Chapter 5 we track source code changes further and categorize them as implementation (new features) or different kinds of maintenance.

Testing can be tracked by extracting changes to unit tests, benchmarks, test code, and test files. Chapter 4 describes how we recover test changes and track them.

Tracking maintenance is important as there are different kinds of maintenance tasks done for different kinds of purposes. These purposes are often dependant on the process being followed. In Chapter 5 we investigate the breakdown of revision by maintenance classifications such as Swanson's maintenance classification. Chapter 6 goes further to see if we can automatically learn from the descriptions of changes and related files in order to classify changes by their maintenance categories.

These different maintenance streams are also integrated into Chapter 8, as the timeline overviews rely on these classification efforts in order to track implementation and maintenance changes across time.

While implementation and maintenance disciplines are the largest contributors of events to the repositories we have, there are many other issues facing a project that are very process oriented, such as deployment and project management.

3.3.3 Deployment, Project Management, and Quality Assurance

Deployment is the packaging, distributing, configuring and executing of a software project. Project management is the management and application of software development processes used to manage a software project, as well as measuring aspects of development. Quality assurance is the attempt to ensure that a software project works, works well and is of a reasonable quality.

How do we extract quality assurance related events and issues from a project? One method is to look for the mention of issues related to software qualities, specifically software quality related such as non-functional requirements like efficiency or reliability. Chapter 7 discusses extracting and labelling developer topics by the non-functional requirement they are related to. Quality assurance is often used to decide if a project is ready to deploy or not.

Deployment can be studied by investigating releases; this is because releases are primarily milestones of a project when a project is complete enough to be packaged and distributed. Chapter 4 studies releases and their patterns in depth. Deployment is also investigated in 8 where we track deployment related events. The timing of deployment is often related how a project is being managed.

Project management is the management of the development of a software project. Project management is hard to track based on the evidence available to us. Unfortunately what is often left behind is just the crumbs, the metrics about a project. Project management also tracks implementation of requirements. While most of these chapters deal with some aspect of project management Chapter 8 explicits seeks to track project management related issues by tracking project management discussions in mailing lists, bug trackers and source control systems.

While deployment and project management events are infrequent they are quite important to software process recovery because they are process-heavy events and actions. Deployment, quality assurance, and project management indicate a conscious effort to apply software development processes to a project.

3.4 Summary

In the following chapters we demonstrate that based upon the evidence available to us — the artifacts of development — we can infer the behaviour and purpose behind many software development events. This inference allows us to recover some software development processes based on the evidence provided. Thus we show that observable software development processes exist within some software projects and that we are capable of tracking some aspects of these processes. We show that observable processes, processes recovered based on evidence, are not clean and discrete like the abstract waterfall model but a mixture of concurrent disciplines and workflows as suggested by the Unified Process. We show that workflows are concurrent and parallel, that the regimented stages, as suggested by the waterfall model, are not applicable for the projects that we studied. We investigate what events and changes occur within these repositories and automatically classify the purpose of many of these events. In Chapter 8 we integrate the research from many of the previous chapters in order to provide a global overview of the underlying software development process.

Chapter 4

Evidence of Process

Within this chapter we demonstrate that we can observe software development processes based on the evidence stored within source control systems. Specifically we derive developer behaviour from the commits and revisions. This chapter also demonstrates that artifacts can be related to concurrent workflows such as implementation and testing.

We will demonstrate that we can observe behaviour and derive de facto software development processes by a simple method of analyzing revisions in source control systems by their file types or purposes. Thus this chapter focuses on the evidence of behaviour allowing us to infer software processes based on behaviours related to development artifacts found in source control systems.

Concurrent development is observable. We show that throughout the life of a project, changes are made to different kinds of artifacts. This shows that changes to artifacts, such as documentation, are not localized to one point in time, in fact as the system evolves artifacts, such as documentation, evolve with the project. We demonstrate that concurrent workflows exist within a project by showing the parallel behaviour of changes to different kinds of artifacts.

Implementation and maintenance changes are observable. With respect to software development processes this chapter helps show that we can observe implementation and maintenance related changes and behaviours. For instance, changes that modify build files or add source code often add new features.

Effectively we leverage multilevel modelling in order to analyze releases of a project. By doing so we improve our ability to describe the value of the release. Studying the release-time activities of a software project — that is, activities that occur around the time of a major or minor release — can provide insights into both the development processes used and the nature of the system itself.

Although tools rarely record detailed logs of developer behaviour, we can infer release-time activities from available data, such as logs from source control systems, bug trackers, etc. In this chapter, we discuss the results of a case study about mining patterns of release-time behaviour from the source control systems of four open source database systems.

We partitioned the development artifacts into four classes — source code, tests, build files, and documentation — to be able to characterize the behavioural patterns more precisely. We found, for example, that there was consistent behaviour around release time within each of the individual projects; we also found these behaviours did not persist across systems, leading us to hypothesize that the four projects follow different but consistent development behaviour around releases.[1]

4.1 Release Patterns

In this chapter we attempt to discover *release patterns*, that is, behavioural patterns of software projects that can be observed around the time of a release. We theorize that release patterns constitute a discernible slice of larger-scale patterns concerning developer behaviour, in that they provide evidence of the actual processes and practices followed by the project members. In turn, we expect release patterns to provide the observer with useful and accurate information about the particular release-time processes being followed. We observe and extract these patterns from the software artifacts available for use. Since many of the development decisions and behaviours are not regularly logged, we rely on systems that automatically log activity, such as a project's source control system (SCS).

Ever since the Cathedral and Bazaar [133], there has been interest from businesses, developers, managers, and researchers about how Free/Libre Open Source Software (FLOSS) is created. Previous attempts at investigating the development processes of Free/Libre Open Source Software (FLOSS) have analyzed Bugzilla repositories and mailing lists [45]; these artifacts do not offer the fine granularity of activity records that revisions, from a SCS, provide. We hope to extract this behaviour so that stakeholders in the project have a way to extract and analyze the behaviour that is reflected within the source control system.

Given that many software processes are composed of stages we suspect we can better observe a system's behaviour by looking for changes or revisions that are related to stages of software development. For example maintenance and implementation stages are related to changes in source code files; integration and test stages are related to changes of test files, benchmarks, test framework code, build files and configuration management files; requirements, specification and design stages are related to changes in documentation files.

[1]This chapter is derived from Release Pattern Discovery via Partitioning: Methodology and Case Study published at MSR 2007 [75] and Release Pattern Discovery: A Case Study of Database Systems, published at ICSM 2007 [68]. Both co-authored with Michael W. Godfrey and Richard C. Holt.

In this research we will study the release patterns of several Open Source Software Database Systems (RDBMSs). Many FLOSS RDBMSs started as proprietary software either commercially or academically. Each has a long history of use and change. We have chosen the domain of database systems, as we believe it to be both mature and fairly stable, and we expect that the individual systems will likely have similar architectures or provide a similar functionality which might help us compare them.

This work, when incorporated into a more general framework, can help people such as managers and programmers. It supports managers in analyzing project behaviour around events such as releases. It supports the extraction of development process information from projects, and so aids managers in verifying what practices the programmers are following. This also permits managers a freer hand in the project development, since it frees them from having to keep a close watch on what the developers are actually doing day-to-day. For example our work can be used to determine when documentation or testing occurs around an event. One could also determine if activities such as programming and testing occur at the same time.

Programmers could investigate how their project is being maintained. Newcomers to the project could determine the process followed by the programmers and figure out the work flow of the project. Programmers could also ask the repository, relative to events, when documentation took place.

Researchers could benefit by correlating the behaviour of successful projects, there by deriving successful software development processes. Researchers could also validate the behaviours of developers against the software development process that the developers claimed they were following.

This work does not analyze all of the project's behaviour. It only analyzes the project's behaviour around release time. In this chapter we look explicitly for common behaviours among all of the case study projects. We conjectured that the release-time practices of these database systems might be similar since they share the same domain.

4.1.1 Background

The *stages* of software development we want to identify are derived from various software development models such as the *waterfall model* [138], the *spiral model* [20] and OMG's *unified process* [86]. Example stages include: requirements, design, implementation, documentation, testing, release, etc.

Software evolution is the study of how software changes over time [48] based on the artifacts left behind by developers. These artifacts include mailing lists, change-logs, program releases, source code, source control systems, revisions, etc. These artifacts often need to be measured or aggregated to be studied. Common software evolution metrics include lines of code, clean lines of code (no comments or extra white-space), number of lines changed, lines added, lines removed, etc. Some software evolution metrics measure systems before and after a change, as well as measuring change itself [121, 115, 48].

4.1.2 Terminology

This section introduces terminology that we use throughout the rest of this chapter.

Source control systems (SCSs) track and maintain the development and revision history of a project. SCSs are repositories which store revisions to files such as source code and documentation. In this study we used CVS and BitKeeper.

Revisions are changes to files stored in a SCS. They are not the actual files themselves but the changes that occur between versions of a file.

Commits are the actions taken that add, submit or record revisions to the SCS.

Releases are the events where software is bundled for distribution. Usually when a version of the software in SCS is decided to be the release, it is checked out and packaged. There are two main kinds of releases: *major releases* and *minor releases*.

Major Releases are releases which are large changes to the software that often affect the software's architecture. Generally developers will indicate a major release by using a large change in their version numbering of a release (e.g. MS Windows 95 to MS Windows 98 or Linux Kernel 2.2 to 2.4).

Minor Releases are generally smaller than major releases and are usually indicated by a smaller change in the version number of a release (e.g. MS Windows XP SP1 to MS Windows XP SP2 or Linux Kernel 2.4.19 to 2.4.20). We note that the criteria used to distinguish between major and minor releases depends on the project.

All releases include both major and minor releases.

Release revisions are revisions that are near a release (e.g., with one week). Given an interval around a release, a release revision is a revision that occurs within that interval.

A partition is one of the four sets of files stored in a SCS that we analyze: *source*, *test*, *build*, and *documentation*.

Revision classes are the sets of revisions to files in our file partitions (source, tests, etc.).

Source revisions are revisions to source code files. Source code files are identified by the file name suffixes such as `.c, .C, .cpp, .h, .m, .ml, .java,` etc. Note that source files might include files which are also used for testing.

Project	Major	Minor	All
Firebird	5	5	10
MaxDB 7.500	1	12	13
MaxDB 7.600	2	11	12
MySQL 3.23	2	68	70
MySQL 4.0	4	110	114
MySQL 4.1	4	110	114
MySQL 5.0	4	110	114
MySQL 5.1	4	110	114
PostgreSQL	7	27	34
Total	33	563	595

Table 4.1: Total number of releases per project (Note MaxDB and MySQL fork their repositories so each repository contains most of the releases)

Test revisions are revisions to files that are used for regression tests, unit tests, etc. Revisions to files that are part of regression test and unit test cases are considered to be test revisions. Generally, any file that has `test` in its name is assumed to be a test file, although there are obvious exceptions.

Build revisions are revisions to build files such as those related with GNU Auto-tools (make, configure, automake, etc) and other build utilities. Common build files have names such as configure, Makefile, automake, config.status, or suffixes such as `.m4`, etc.

Documentation revisions are revisions to documentation files, which include files such as README, INSTALL, doxygen files, API documents, and manuals.

Project behaviour is the behaviour of the source, test, build and documentation revisions around an event, based on the project's SCS.

Release patterns are the patterns of project behaviour that are observed around a release. A *release pattern* is a behaviour that occurs before, after or during a release. A release pattern includes behaviours such as increased frequency of documentation revisions before a release which drop off after the release, or even the frequency of test revisions maintaining a constant rate during the release. These patterns are primarily found by analyzing a project's release revisions.

4.2 Methodology

This section presents our methodology for analyzing release patterns of a project; we will present the steps involved and then we will followup with an application of our methodology in a case study (Section 4.3).

Project	Source	Test	Build	Doc
Firebird	40737	7727	3183	534
MaxDB 7.500	10369	4270	298	52
MaxDB 7.600	23456	7087	318	97
MySQL 3.23	4220	1410	421	21
MySQL 4.0	11593	4936	1033	34
MySQL 4.1	31451	16430	2990	88
MySQL 5.0	45946	26373	3908	105
MySQL 5.1	52897	31389	4772	122
PostgreSQL	39153	4906	7172	3084
Total	259822	104528	24095	4137

Table 4.2: Total Number of Revisions per class per project

Our project analysis methodology can be summarized as: extracting data for revisions and releases; partitioning the revisions into their revision classes; grouping revisions by aggregation and windowing; producing plots and tables; analyzing summaries of the results with our STBD notation (see Section 4.2.5).

Our project analysis methodology can be summarized as:

- Extracting data for revisions and releases (Section 4.2.1);
- Partitioning the revisions (Section 4.2.2);
- Grouping revisions by aggregation and windowing (Section 4.2.3);
- Producing plots and tables (Section 4.2.4);
- Analyzing summaries of the results with our STBD notation (see Section 4.2.5).

4.2.1 Extraction

First, we choose a target project's SCS and either mirror the repository or download each revision individually. From SCSs such as CVS or BitKeeper we extract the revisions and sometimes release information. We will later analyze this extracted data. Per each revision the minimal information extracted includes the date of revision, the name of the revised file and the author of the revision. In Section 4.3.2 we describe our extractors: `softChange` for CVS repositories and `bt2csv` for BitKeeper repositories.

Releases are found manually by evaluating mailing lists SCS tags, project change-logs, manuals, and even the release date-stamps found in the project's FTP repository. Once extraction is complete we are ready to partition our revisions into classes. The release information we extract includes the version number, the date, and whether the release is a major release or not.

4.2.2 Partitioning

Once we have extracted the revisions, we partition the set of revisions based on their of revised files into these four classes: source code, tests, build scripts and documentation. In principle, these are disjoint sets, but in the work presented here, there is some overlap between source code and tests.

We partition the revisions into their respective classes by suffix and if their names match. Usually test files are classified as test because they are in a test directory, they have "test" in their pathname, or they have a test related prefix or suffix. Documentation files are determined much the same way. Suffixes help determine source files and build files. One should audit the matched files and determine which ones truly belong to each class, as there are sometimes false positives. If revisions are duplicated between branches, we will evaluate each duplicate as a separate revision. We do not filter out revisions except for those which do not fit into any revision class.

4.2.3 Aggregates

Our revision data is often quite variable and produces messy plots, thus aggregation and smoothing of revision data can be used to get a clearer picture of the underlying trends. For instance, revision frequency data often follows a Pareto or power-law distribution, meaning that points are highly variant, and there are lot of points that look like outliers but are a normal part of development.

Smoothing can be achieved using aggregate functions such as summation over an interval, summation over a window, average over a window, etc.

We have a choice between aggregating aggregates, averaging aggregates or combining all the revisions and aggregating them together. An example of this is to group all of the revisions that occurred 1 week before a release together and analyze those results, the alternative would be to analyze each release independently of each other and then aggregate these results.

In summary we aggregated and grouped our revision class partitions for plotting and analysis. Aggregates include averages, summations and windowed functions. In this study we aggregated the revisions by day and filtered the revisions into per-day bins before and after release for a given interval.

4.2.4 Analysis

Using the aggregated and partitioned data we then plot various graphs such as revision frequency before and after a release, and the linear regression of changes before and after a release. From these plots we generate summaries of the slopes of the linear regression or other aspects of the graphs themselves. These summaries often use the STBD notation which we describe in detail in Section 4.2.5. We then analyze the summaries and discover release patterns from the data and our summaries.

Once these steps are done, for each project analyzed we compare and contrast the summaries of the projects to determine if they have a similar relative behaviour.

We use graphs, plots and tables to help us understand the release patterns of a project. These help us answer the question: *For each class of file, does the frequency of revision increase (or decrease) preceding (or following) the time of a release?*

To answer a question such as this, we plot the frequency of revisions in a revision class before and after a release. We prepare our data, as suggested in the previous section, by aggregating by a time period, such as hours or days. We then would compare the average of the number of revisions in each period. We would do this multiple times with different values for parameters such as interval length or release type. *Interval length* refers to the time, the window, around a release that we are analyzing. We can generate plots of the revision frequencies and linear regressions of these frequencies. We can also make tables that show how the results change when parameters like interval length change. If we have too many results we might want to compound these results by aggregation functions like majority voting or averages to make them more readable and analyzable.

In our experiments, as described in our case study, we varied the length of the interval, around the releases, from 7 days up to 42 days. We carried out linear regressions on the aggregated frequencies for the before release and after release intervals to help identify release patterns.

Given the previous steps in our methodology and given the resulting plots and tables, the final step is to analyze these tables and plots to help us understand the release patterns. Our greater goal is to gain insight about the process of software change. As we have mentioned this often requires us to summarize the behaviour of the attributes and metrics of graphs and tables such that we can make general claims about behaviour in a verifiable and definitive manner. We developed a summary notation which we call STBD notation, described below in Section 4.2.5, to help summarize trends of revisions before, after and during releases.

4.2.5 STBD Notation

STBD notation is a short form summary of the results of a query much like Myers-Briggs Type Indicator (MBTI) [123]. MBTIs are short form summaries which show the preference of the individual for each of the four dichotomies which are represented positionally in order by single characters: Extroversion and Introversion (E/I), Sensing and iNtuition (S/N), Thinking and Feeling (T/F), Judging and Perceiving (J/P). Example MBTIs include INTJ and INTP.

STBD notation is similar to MTBI, but is meant to summarize comparisons and representations of values related to revision classes. STBD notation is a notation to summarize the results of metrics on our four classes of revisions. We assign a letter to each revision class (S for source, T for test, B for build, D for documentation). We order the class letters from most frequent class to least frequent class: `S`, `T`, `B`, `D`.

The format of the summary is `S*T*B*D*` where `*` could be characters such as ▼ (down/before), □(equals), ▲ (up/after). Generally we try to choose intuitive mappings for these characters, for instance ▼, ▲, and □ map well to the slope of a line (such as the linear regression of a set of points). ▲ and ▼ work well for comparing two values, much like less than and greater than. In this chapter when we compare two metrics such as revision frequency before and after a release, ▼ means the revision frequency was greater before a release, and ▲ means the revision frequency was greater after a release.

An example instance of STBD notation would be, `S▼T▼B▲D□` for revision frequency before and after a release; this would indicate source and test revisions were more frequent before the release, `S▼T▼`, while build revisions were less frequent before, `B▲`, and documentation revisions were equally frequent, `D□`. We can also omit classes that we are not interested in or did not change. For example if only source revisions changed, or we only want to focus on source revisions, we could show only `S▲`.

This notation is flexible and allows us to identify patterns and make quick judgements based on actual data, the repetition of the class letters helps readers avoid memorizing the ordering. Metrics we can use with STBD notation include: linear regression slopes, average LOC per revision, average frequency of revision, relative comparison of frequencies, the sign of a metric, concavity of quadratic regression, etc.

4.3 Case Study

In this case study we study four FLOSS databases: PostgreSQL, MySQL, Firebird and MaxDB. We start with initial questions and predictions. We then apply our methodology, as discussed in Section 4.2, and then analyze and compare these projects.

Project	Major	Minor	All
Firebird	S▼T▲B▼D▲	S▼T▼B▲D▼	S▼T▲B▼D▲
MaxDB	S$\overset{?}{\square}$T▼B▼D$\overset{?}{\square}$	S$\overset{?}{\square}$T▲B$\overset{?}{\square}$D$\overset{?}{\square}$	S$\overset{?}{\square}$T$\overset{?}{\square}$B$\overset{?}{\square}$D▼
MySQL	S▼T▼B▲D▼	S▼T▼B$\overset{?}{\square}$D▼	S▼T▼B$\overset{?}{\square}$D▼
PostgreSQL	S▲T▲B▲D▲	S▲T▼B▼D▲	S▲T▼B▲D▲

Table 4.3: Summary of the four major project's revision frequencies before and after a release with majority voting across the branches, where $\overset{?}{\square}$ is when no majority is found, ▼ means revisions are more frequent before, ▲ means revisions are more frequent after. S - source, T - test, B - build, D - documentation

4.3.1 Questions and Predictions

For each revision class, we ask these questions:

- Are revisions of this class more frequent before or after a release?
- Is the frequency of change increasing or decreasing before or after a release?

As a general trend, we expected most projects to obey S▼T▼. That is, we expect to see more code and testing added just before a release than just after. We also expected few if any build changes just before a release, as they usually correspond to the addition of new — and therefore risky — features in code. That is, we expected most projects to obey B□ or B▲.

4.3.2 Tools and Data-sets

We used a set of extractors on various data-sets and then analyzed the extracted data with our analysis tools.

Extractors we used include: *CVSSuck* is a tool that mirrors RCS files from a CVS repository; ***softChange*** extracts CVS facts to a PostgreSQL database; *bt2csv* converts BitKeeper repositories to facts in CSV databases.

Data-sets we used include: *PostgreSQL* (CVS), *Firebird* (CVS), *MaxDB* 7.500, 7.600 (CVS), *MySQL* 3.23, 4.0, 4.1, 5.0, 5.1 (BitKeeper). Summaries of the number of releases and revisions per class are shown in tables 4.1 and 4.2.

Analysis tools we used include: *Hiraldo-Grok* (an OCaml based variant of Grok used for answering queries), *R* (a plotting and Statistics Package), *GNUplot* (a graph plotting package), and *Octave* (an FLOSS Matlab clone).

Project	Major	7 days	14 days	31 days	42 days
Firebird	Major	S▲T▼B▼D▲	S▼T▲B▼D▼	S▼T▲B▼D▲	S▼T▲B▼D▲
MaxDB 7.500	Major	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MaxDB 7.600	Major	S▲T▲B▼D□	S▲T▼B▼D▼	S▲T▼B▲D▲	S▼T▼B▼D□
MySQL 3.23	Major	S▼T▲B▲D□	S▲T▼B▲D▼	S▲T▼B▲D▼	S▲T▼B▲D▼
MySQL 4.0	Major	S▼T▼B▲D□	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▲D▼
MySQL 4.1	Major	S▼T▲B▲D□	S▼T▼B▲D▼	S▼T▼B▲D□	S▼T▼B▲D▲
MySQL 5.0	Major	S▼T▼B▲D▼	S▼T▼B▲D▼	S▲T▼B▲D▼	S▼T▼B▲D▲
MySQL 5.1	Major	S▼T▼B▲D▼	S▼T▼B▲D▼	S▲T▼B▲D▼	S▼T▼B▲D▲
PostgreSQL	Major	S▼T▲B▲D▼	S▲T▲B▲D▲	S▲T▲B▲D▲	S▲T▲B▲D▲
Project	Major	7 days	14 days	31 days	42 days
Firebird	Minor	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▲D▼
MaxDB 7.500	Minor	S▼T▲B▼D▲	S▼T▲B▼D▲	S▲T▲B▼D▲	S▲T▲B▲D▲
MaxDB 7.600	Minor	S▲T▲B▲D▼	S▲T▲B▲D▼	S▲T▲B▲D▲	S▲T▲B▲D▼
MySQL 3.23	Minor	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MySQL 4.0	Minor	S▼T▼B▲D▼	S▼T▲B▲D▼	S▼T▼B▼D▼	S▼T▲B▼D▼
MySQL 4.1	Minor	S▼T▼B▼D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▼D▼
MySQL 5.0	Minor	S▼T▼B▼D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▼D▼
MySQL 5.1	Minor	S▼T▼B▼D▼	S▼T▲B▲D▼	S▼T▲B▼D▼	S▼T▲B▼D▼
PostgreSQL	Minor	S▼T▼B▼D▲	S▲T▲B▲D▲	S▲T▼B▼D▲	S▲T▼B▼D▲
Project	Major	7 days	14 days	31 days	42 days
Firebird	All	S▼T▼B▼D▲	S▼T▲B▼D▼	S▼T▲B▼D▲	S▼T▲B▼D▲
MaxDB 7.500	All	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼	S▲T▲B▼D▼
MaxDB 7.600	All	S▲T▲B▼D▼	S▲T▲B▲D▼	S▲T▲B▲D▲	S▼T▼B▼D▼
MySQL 3.23	All	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MySQL 4.0	All	S▼T▼B▲D▼	S▼T▲B▲D▼	S▼T▼B▼D▼	S▼T▲B▼D▼
MySQL 4.1	All	S▼T▼B▼D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▼D▼
MySQL 5.0	All	S▼T▼B▼D▼	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▼D▼
MySQL 5.1	All	S▼T▼B▼D▼	S▼T▲B▲D▼	S▼T▲B▲D▼	S▼T▲B▼D▼
PostgreSQL	All	S▼T▼B▼D▲	S▲T▲B▲D▲	S▲T▼B▲D▲	S▲T▼B▲D▲

Table 4.4: A STBD notation summary table of project revision frequencies across release types, and interval lengths. ▼ means revisions are more frequent before a release, ▲ means revisions are more frequent after a release. □ means revisions were equally frequent before and after a release. S - source, T - test, B - build, D - documentation

Project	Major	Minor	All
Firebird	S▼T▲B▼D▲	S▼T▼B▲D▼	S▼T▲B▼D▲
MaxDB 7.500	S▼T▼B▼D▼	S⍰T▲B▼D▲	S▼T▼B▼D▼
MaxDB 7.600	S▲T▼B▼D□	S▲T▲B▲D▼	S▲T▲B⍰D▼
MySQL 3.23	S▲T▼B▲D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MySQL 4.0	S▼T▼B▲D▼	S▼T⍰B⍰D▼	S▼T⍰B⍰D▼
MySQL 4.1	S▼T▼B▲D□	S▼T▼B⍰D▼	S▼T▼B⍰D▼
MySQL 5.0	S▼T▼B▲D▼	S▼T▼B⍰D▼	S▼T▼B⍰D▼
MySQL 5.1	S▼T▼B▲D▼	S▼T▲B▼D▼	S▼T▲B⍰D▼
PostgreSQL	S▲T▲B▲D▲	S▲T▼B▼D▲	S▲T▼B▲D▲

Table 4.5: Summary of Table 4.4 using majority voting where ⍰ means no majority. ▼ means revisions are more frequent before a release, ▲ means revisions are more frequent after a release. □ means revisions were equally frequent before and after a release. S - source, T - test, B - build, D - documentation

We selected four successful FLOSS RDBMSs to base our study on. Two of the RDBMSs originally started in universities (PostgreSQL and MySQL) while two were gifted from commercial companies to the Open Source Community (Firebird and MaxDB). Of those RDBMSs, we used multiple forks of the databases (MySQL and MaxDB) primarily because their major release branches did not share the same SCS repositories.

The database domain was chosen for study as instances of this domain implement highly similar functionality, and rely on a large body of well known ideas and experience. That is, we expected them to have some architectural traits in common, at least at the coarse grained level, and this makes them good candidates for comparison. Additionally, these projects are relatively mature and thus have automated tests, benchmarks and developer documentation, so there are many kinds of artifacts to study.

4.4 Results

In this section we show and discuss our results from many different viewpoints. We measured the revision frequency and change of the revision frequency of our four revision classes for each of the four different RDBMS. We then analyzed these measurements from multiple perspectives.

We have plotted much of the data and provided it in a summarized form. We evaluated this data from various perspectives and viewpoints such as release types, interval length, project, revision classes, and the linear regression of the frequencies of revision classes. We also evaluate our assumptions and discuss the validity threats that face our analysis.

Project	Major	Before	After	Both
Firebird	Major	S▲T▼B▲D▼	S▼T▲B▼D▼	S▲T▼B▲D▼
MaxDB 7.500	Major	S▲T▲B▼D▲	S▼T▼B□D□	S▲T▲B▲D▲
MaxDB 7.600	Major	S▲T▲B▲D▲	S▲T▲B▲D▼	S▲T▲B▲D▲
MySQL 3.23	Major	S▲T▲B▼D▼	S▼T▲B▼D□	S▼T▲B▼D▼
MySQL 4.0	Major	S▼T▲B▲D▲	S▼T▲B▼D□	S▼T▲B▼D▲
MySQL 4.1	Major	S▼T▲B▲D▼	S▲T▲B▼D▼	S▲T▲B▼D▼
MySQL 5.0	Major	S▼T▲B▲D▲	S▲T▲B▼D▲	S▲T▲B▼D▼
MySQL 5.1	Major	S▼T▲B▲D▲	S▼T▲B▲D▼	S▼T▲B▼D▼
PostgreSQL	Major	S▲T▼B▲D▲	S▼T▲B▼D▼	S▼T▲B▼D▼

Project	Major	Before	After	Both
Firebird	Minor	S▼T▲B▲D▲	S▲T▼B▼D▲	S▼T▲B▼D▲
MaxDB 7.500	Minor	S▲T▲B▼D▲	S▼T▼B▼D▼	S▼T▼B▼D▼
MaxDB 7.600	Minor	S▲T▲B▲D▲	S▲T▲B▲D▼	S▲T▲B▲D▼
MySQL 3.23	Minor	S▼T▼B▲D▼	S▼T▼B▲D▼	S▲T▼B▲D▼
MySQL 4.0	Minor	S▼T▼B▲D▼	S▼T▼B▲D▼	S▼T▼B▲D▲
MySQL 4.1	Minor	S▼T▼B▲D▲	S▲T▲B▲D▼	S▲T▼B▲D▲
MySQL 5.0	Minor	S▼T▼B▲D▲	S▲T▲B▲D▼	S▲T▼B▲D▲
MySQL 5.1	Minor	S▼T▼B▲D▲	S▲T▲B▲D▼	S▲T▼B▲D▲
PostgreSQL	Minor	S▼T▼B▼D▲	S▲T▲B▲D▲	S▼T▲B▲D▼

Project	Major	Before	After	Both
Firebird	All	S▼T▼B▲D▼	S▼T▲B▼D▼	S▼T▼B▲D▼
MaxDB 7.500	All	S▼T▼B▼D▼	S▼T▼B▼D▼	S▼T▼B▼D▼
MaxDB 7.600	All	S▲T▲B▲D▲	S▲T▲B▲D▼	S▲T▲B▲D▲
MySQL 3.23	All	S▼T▼B▲D▼	S▼T▼B▼D▼	S▲T▲B▲D▼
MySQL 4.0	All	S▼T▼B▲D▼	S▼T▼B▼D▼	S▼T▼B▲D▲
MySQL 4.1	All	S▼T▼B▲D▲	S▲T▲B▲D▼	S▲T▲B▲D▲
MySQL 5.0	All	S▼T▼B▲D▲	S▲T▲B▲D▼	S▲T▲B▲D▲
MySQL 5.1	All	S▼T▼B▲D▲	S▲T▲B▲D▼	S▲T▼B▲D▲
PostgreSQL	All	S▼T▼B▼D▲	S▲T▲B▲D▲	S▼T▲B▼D▼

Table 4.6: Linear Regressions of daily revisions class totals (42 day interval): ▼ indicates a positive slope, ▲ indicates a negative slope, □ indicates a slope of 0

Project	Major	7 days	14 days	31 days	42 days
Firebird	Major	▲(0.56)	▲(0.65)	▼(0.50)	▼(0.50)
MaxDB 7.500	Major	▼(0.01)	▼(0.01)	▼(0.01)	▼(0.03)
MaxDB 7.600	Major	▲(0.60)	▲(0.58)	▲(0.53)	▼(0.28)
MySQL 3.23	Major	▼(0.42)	▼(0.32)	▼(0.36)	▼(0.34)
MySQL 4.0	Major	▼(0.40)	▼(0.35)	▼(0.36)	▼(0.36)
MySQL 4.1	Major	▼(0.48)	▼(0.46)	▼(0.45)	▼(0.41)
MySQL 5.0	Major	▼(0.45)	▼(0.47)	▼(0.48)	▼(0.45)
MySQL 5.1	Major	▼(0.45)	▼(0.44)	▼(0.49)	▼(0.46)
PostgreSQL	Major	▼(0.46)	▲(0.64)	▲(0.61)	▲(0.62)
Project	Major	7 days	14 days	31 days	42 days
Firebird	Minor	▼(0.30)	▼(0.37)	▼(0.38)	▼(0.44)
MaxDB 7.500	Minor	▼(0.49)	▲(0.50)	▲(0.72)	▲(0.83)
MaxDB 7.600	Minor	▲(0.66)	▲(0.66)	▲(0.57)	▲(0.51)
MySQL 3.23	Minor	▼(0.31)	▼(0.37)	▼(0.41)	▼(0.43)
MySQL 4.0	Minor	▼(0.45)	▼(0.47)	▼(0.46)	▼(0.48)
MySQL 4.1	Minor	▼(0.45)	▼(0.48)	▼(0.48)	▼(0.48)
MySQL 5.0	Minor	▼(0.47)	▼(0.48)	▼(0.48)	▼(0.48)
MySQL 5.1	Minor	▼(0.48)	▼(0.49)	▼(0.50)	▼(0.50)
PostgreSQL	Minor	▼(0.40)	▲(0.59)	▲(0.52)	▲(0.50)
Project	Major	7 days	14 days	31 days	42 days
Firebird	All	▼(0.37)	▲(0.51)	▼(0.45)	▼(0.48)
MaxDB 7.500	All	▼(0.06)	▼(0.10)	▼(0.32)	▲(0.55)
MaxDB 7.600	All	▲(0.65)	▲(0.65)	▲(0.57)	▼(0.47)
MySQL 3.23	All	▼(0.32)	▼(0.36)	▼(0.40)	▼(0.43)
MySQL 4.0	All	▼(0.45)	▼(0.47)	▼(0.46)	▼(0.48)
MySQL 4.1	All	▼(0.45)	▼(0.48)	▼(0.48)	▼(0.48)
MySQL 5.0	All	▼(0.47)	▼(0.48)	▼(0.48)	▼(0.48)
MySQL 5.1	All	▼(0.48)	▼(0.49)	▼(0.49)	▼(0.49)
PostgreSQL	All	▼(0.41)	▲(0.60)	▲(0.54)	▲(0.53)

Table 4.7: Comparison of average of total number of revisions before and after a release. ▼ indicates more revisions before a release, ▲ indicates more revisions after a release.

4.4.1 Indicators of Process

Our results indicate that there was no global consistency across revision classes, across all projects, but that there was some consistency within the individual projects (see Table 4.4). This consistency suggests that each of the projects observed regular patterns of behaviours — that is, they were following a *process* — but that different projects followed different processes.

One indicator of process that seemed extractable was how test revisions could relate to the test methodology used by the project. Looking at MySQL 3.23 to 4.1 we can see that source revision activity increased across the release (Table 4.6) while test revision activity decreased (it had a negative slope after release as well). This suggests that source revisions were not heavily correlated with test revisions. PostgreSQL and Firebird's test revisions seemed to change in the opposite way of their source and build revisions. MaxDB, on the other hand, had correlated release patterns between source and test revisions. This suggests that MySQL, Firebird, and PostgreSQL did not follow a test-driven development model, or that testing was done at a different time in the process, yet MaxDB followed a release time process or pattern in where test changes and source changes are correlated.

4.4.2 Linear Regression Perspective

We used the STBD notation to describe the results of multiple linear regressions of revisions per day before, after and during a release (Table 4.6) with before and after intervals of 42 days.

Notable release patterns recovered from the linear regressions were: `S▼T▼`, source and test changes sloping downward, for the before release behaviours were common for *All* releases for every project except for MaxDB 7.6; a slope of `B▼` was common among *before* intervals for *Minor* releases and *All* releases.

In Table 4.6, we can see that *after* release summaries for *All* and *Minor releases* were very similar with the most notable change in build files for MySQL. *Before* release patterns of `S▲T▲` were consistent with after patterns of `S▼T▼` for MySQL 4.1 to 5.1 and PostgreSQL. This indicates there was a concave up peak around release time. Peaks and dips around release time are interesting because they seem to indicate that an event that changes developer behaviour occurred.

In the *Both* interval the most consistent pattern was `B▼` for *Minor* and *All releases* and the opposite for *Major* releases (`B▲`). The `B▲` pattern matches with the summaries in Table 4.5. For all releases and all intervals, except *Major After* and *Major Both*, we observed `B▼` for build revisions. This suggests that build revisions are probably less common around release time than during the non-release time. We observed there was almost a freeze in build changes around a *Major* release as if features which include new files or compilation changes were being held back.

4.4.3 Release Perspective

We can see from Table 4.5 and Table 4.3 that there was a definite difference in behaviour between *Major* and *Minor* releases per each project and across projects. These tables summarize the average frequency before and after release. Source revision behaviour seemed relatively consistent between *Major* and *Minor* releases, except for MySQL 3.23 and MaxDB 7.500.

Test behaviour was inconsistent across all projects other than MySQL 3.23, 4.1 and 5.0. Test release patterns were different between *Major* and *Minor* releases. For Firebird, MaxDB and PostgreSQL, *Major* and *Minor* test patterns were completely opposite to each other. Firebird and PostgreSQL modified tests more frequently before a *Minor* release than after, where as for MaxDB and for all the MySQLs tested more often before a *Major* release than after; yet for *Minor* releases test release patterns were the opposite of MaxDB and were inconsistent for MySQL.

Projects with inconsistent build release patterns were Firebird, MaxDB 7.600, MySQL and PostgreSQL. Only Firebird and MaxDB had `B`▼for *Major* releases yet mostly `B`▲ for *Minor* releases. MySQL and PostgreSQL seemed to change build files more before *Minor* releases rather than *Major* releases.

A general release pattern of PostgreSQL, that was consistent across all releases, was that there were more revisions to PostgreSQL after a release than before, except for the one week interval. Perhaps this indicates that PostgreSQL follows a process that relies on code freezes, or a delay of patches till after a release.

The differences between *Major* and *Minor* releases seemed similar between projects but were mostly noticeable in MaxDB 7.5. The differences between *Major* and *Minor* releases seemed more prevalent than differences between intervals.

4.4.4 Interval Length Perspective

Tables 4.4, and 4.7 both provide summaries of data organized by interval. The one week interval in Table 4.4 showed that *Major* and *Minor* releases seem to act quite differently, but when combined, *All* releases had more changes before release than after. The week *before* release generally had more activity than the week *after*. We can confirm this in table 4.7, where we see only for one release of MaxDB and the *Major* releases of Firebird, were there more revisions during the week after release than before. Yet for *Major* releases of projects such as MySQL and PostgreSQL there were more build revisions after a release than before.

Another interesting release pattern related to one week intervals was that documentation had the most equally frequent results (MaxDB 7.6 and MySQL 3.23 to MySQL 4.1). These equally frequent results suggest there were not a lot of revisions to documentation during the *Major* releases (alternatively, the behaviour was stable). The longer the interval was, the more noticeable and trackable the documentation revisions became. This was most likely due to the infrequency of documentation revisions.

Some projects have shown some stability across intervals, such as the pattern of PostgreSQL for intervals of 14 days or greater for *Major* releases, and the pattern of MySQL 3.23 for *Minor* and *All* releases, for all intervals plotted in table 4.4. MySQL 3.23 was also somewhat consistent for *Major* releases beyond the one week interval. Some projects went through a slow transition of behaviours from one interval to the next.

This interval based analysis lends itself to per-project analysis since there was inconsistency between projects.

4.4.5 Project Perspective

Probably the most notable perspective is the per-project perspective: the results are generated internally from a project, and show consistency within a project.

At the time of this study in 2007 Firebird did not have a lot of releases and was recently released by Borland to the Open Source community. Firebird's shape for source revisions around *Major* releases was a concave up dip in frequency around release time followed by a rise; for *Minor* releases its source revision frequencies had a concave down shape where the frequency peaks around the release. The opposite behaviour was observed for test revisions. Source revisions were consistently more frequent before a release across all release intervals. It seemed that Firebird was generally documented around the time of *Major* releases rather than *Minor* releases.

The MaxDB branches are particularly interesting because the behaviour between branches is often inconsistent or directly opposite. For *Major* releases MaxDB had consistent revision frequencies of `T▼B▼`. The linear regression of *Major* revisions, for the *both* interval and the *before* interval, were consistent and downward sloping; *Minor* releases inconsistently displayed the opposite behaviour for everything except documentation revisions. For *All* releases of MaxDB 7.5, all intervals and every revision class, except documentation, they had the opposite slope of MaxDB 7.6.MaxDB 7.5 did not have many revisions immediately after release. Perhaps there were not enough MaxDB revisions but the nature of development could have changed from adding new functionality to maintaining the code.

The MySQL branches are not totally consistent but they show a transitioning consistency through their branches. In general MySQL followed S▼T▼D▼ for revision frequency; build revisions followed a B▲pattern for Major releases, but otherwise were often inconsistent. According to Table 4.2 we can see that the proportion of test revisions to source revisions grew to over 5 : 3, therefore testing within MySQL seems very important. Slopes of S▲T▲ before and S▲T▲ after, and S▲T▲ before and S▼T▼ after were observed for MySQL's *Minor* and *All* releases. The shape of the curve for source and test revisions around release time for MySQL was a general upward slope or a concave down peak.

PostgreSQL was the only project to have more build revisions than test revisions. For *Minor* and *All* releases, for all revision classes except documentation, PostgreSQL had a concave down peak at release (e.g. slopes of S▲T▲B▲ to S▼T▼B▼). For *Major* releases, PostgreSQL had a concave up dip for source, build and documentation revisions and had a concave down shape for test revisions. For the *both* interval, there was a positive slope for all revision classes except for tests. PostgreSQL seemed to have the most emphasis on frequent change after release; PostgreSQL's behaviour seemed most indicative of the kind of project that uses code freezes before release.

4.4.6 Revision Class Perspective

Source revisions were the most common revisions of all the revision classes. For most intervals and all releases (*Major*, *Minor*, *All*) source revisions were usually more frequent before a release than after. Only for MaxDB 7.6 and PostgreSQL were source changes more frequent after a release than before. For all projects, it seems that source revision frequency almost returned to an equilibrium during the 42 day window. Overall, slope of the linear regression of source revisions was inconsistent across *Major* and *Minor* releases. For *All* releases, source revisions have a consistent positive slope before release, the slope usually changed after release.

Test revisions were the second most numerous kind of revision across all of the projects. For *Minor* releases and *All* releases, tests were usually more frequent *before* a release than *after*. For *Minor* releases the slope of the frequency change across the release was positive. *Major* releases had a negatively sloped test revision frequency across all intervals. Whereas across *Minor* and *All* releases, tests had a concave down peak around release time. For Firebird and PostgreSQL, tests were more frequent after a *Major* release, where as for MaxDB and MySQL they were often more frequent before release. For *Minor* releases we observed the opposite behaviour for all projects except MySQL. Tests seemed to be correlated with the frequency of revisions before and after release.

Build revisions were the third most frequent revisions for all projects except PostgreSQL. For *Minor* releases build revisions were negatively sloped across the release where as for *Major* releases they were positively sloped across the release, and dipped down around release time. Perhaps build revisions were more probable after a *Major* release because new functionality was added that required more configuration changes. Build revisions seemed to be inconsistently frequent for all releases, but had a more consistent negative slope across *Minor* releases.

Documentation revisions were the least frequent revisions for all projects except PostgreSQL. The frequency of documentation revisions indicated that they were changed more often before a release rather than after, which suggests that documentation was left until the functionality was frozen so that it could be described. *Major* releases of Firebird, and all releases of PostgreSQL had documentation changes that were more frequent after release than before. Documentation revision frequencies were the most likely to be equal across a release mainly due to the lack of documentation revisions. The shape of documentation revisions around *Minor* releases was usually a dip, but this was inconsistent for *All* and *Major* releases.

Source, test and documentation revisions seemed to be related and often mimicked each other in frequency, but usually not in slope near release time. Build revisions seemed to be the most at odds with the behaviour of other revision classes, and were the most likely to be more frequent after release. Documentation revisions were the most likely to be missing after release because they were the least frequent.

4.4.7 Zipf Alpha Measure

In order to compare the distribution with respect to file change frequency we found closest fit alpha parameter for the Zipf function. We assumed that the closer the source, test, build and documentation alphas are to each other the more similar the distribution. The results of determining the Zipf α that was the most similar (mean squared distance) are shown in table 4.8. These are for all revisions of that class, not just the release revisions.

4.4.8 Answers to Our Questions

This section attempts to address questions asked in previous We answered questions from Section 4.3.1 based on our frequency tables (Tables 4.5 and 4.4). In particular we answered, “For each class of revision, are revisions in that class more frequent before or after release?”

Source revisions, in general, were changed more frequently before a release than after. There were exceptions such as the early MySQLs, MaxDB 7.600 and PostgreSQL. This suggests that the developers are probably finalizing the project for release and fixing as much as possible before the release. Projects that had more source changes after release might save large risky changes until after the release.

Project	Source α	Test α	Build α	Doc α
Firebird	1.3489	0.4373	0.5203	0.8243
MaxDB 7.500	0.4426	0.5841	0.8836	0.5308
MaxDB 7.600	0.5026	0.6323	0.7710	0.6369
MySQL 3.23	0.7352	0.6125	0.6399	0.7173
MySQL 4.0	0.7507	0.6692	0.4719	0.9468
MySQL 4.1	0.7638	0.6842	0.4389	0.9931
MySQL 5.0	0.7887	0.6821	0.4883	1.0753
MySQL 5.1	0.7734	0.6874	0.4857	1.0924
PostgreSQL	0.5349	0.5759	1.5661	0.7931

Table 4.8: The Zipf α parameters for each class of revision.

Test revisions were usually more frequent before either a *Major* or *Minor* release than after. This suggests that testing is often done to verify that a project is ready for release.

Build revisions, for *Major* releases were usually done more after a release than before; the opposite was true for the *Minor* releases for all projects, except for Firebird and MaxDB 7.600. Large changes that would require changes to build files are probably saved for after a major release, as one would expect a major release to stabilize functionality. Where as a minor release might be prompted by adding a new feature.

Documentation revisions were changed more before release than after for MySQL and MaxDB; Firebird and PostgreSQL displayed the opposite behaviour for *Major* releases. Documentation changes are probably related to new functionality, thus since source changes are changed more after a major release for a project, documentation is probably changed at the same time. In general though, one could expect documentation to change more before a release because the functionality has solidified and thus can be documented.

Is the frequency of change increasing or decreasing as we approach a release? To answer this question we rely on the linear regression results from Table 4.6.

Source revision, as shown in Table 4.6, frequency had a consistent before release behaviour (`S`▼) for *Minor* and *All* releases, but was inconsistent for *Major* releases. For *Major* releases only PostgreSQL and some MySQL branches had `S`▲ across the release. Consistent patterns found were: both intervals had `S`▼ (downward slopes), and both intervals had `S`▲ slopes and concave down dips. Source revisions generally increased before a release except with *Major* releases.

Test revision slopes were `T`▼ across *Major* releases, and `T`▲ across *Minor* releases. The slope across *All* releases was inconsistent across the projects.

Build revision slopes for some MySQL and PostgreSQL *Major* releases, increased across the release (`B`▲). PostgreSQL's *Major* release build revisions formed a concave up shape. For Firebird and MaxDB the general trend was `B`▼. For *Minor* and *All* revisions PostgreSQL had a concave down peak around release.

Documentation revision activity increased across the release (`D`▲) for both Firebird and PostgreSQL. After a release, documentation revision frequency often had a flat slope, which indicated that there was no documentation activity after a release for most projects except Firebird, PostgreSQL and MaxDB 7.600.

Thus we can see that due to the differences between projects there was usually no clear consensus on what each revision class does, but per project, as we saw with MaxDB and MySQL, there was some internal consistency between branches.

Overall, we found our prediction of `S`▼ was consistent for all projects except MaxDB 7.600, MySQL 3.23 and PostgreSQL (table 4.5). Also, our prediction of `T`▼ revision frequency was common for both MaxDB databases and all *Major* releases of MySQL, although MaxDB's *Major* releases were inconsistent with their *Minor* and *All* releases. Build revisions before *Major* releases occurred more often after release than before for MySQL and PostgreSQL; this is consistent with the hypothesis that structural changes, such as the addition of new features, were rare just before major releases. However, for *Minor* and *All* releases the general trend was that build revisions were either inconsistent or were more frequent before release. Finally, we found that many of our predictions were not borne out by the data; in particular, we did not expect to find such inconsistency between the projects.

4.4.9 Validity Threats

We studied four Open Source RDBMSs. While these cover all of the major systems in this category, it is not at all clear that any results can be further generalized to other RDMBs or to Open Source systems in general.

Our six main threats to validity were: deciding if a release was a *major* release or a *minor* release; determining if branching seen in MySQL and MaxDB affected our results; determining if we had enough revisions per aggregate to be statistically significant; deciding on an appropriate interval length; comparing the projects against each other based on their internal measurements; whether or not the linear regression is appropriate for time series data such as this. None of the projects were within a single stage of development, it was a mixture of disciplines.

Our interpretation of major and minor releases might have differed from the developer's interpretation. For instance, we did not assume the internal MySQL 4.0.0 release was a major release, we instead labelled the successive public release as the major release [75].

The MySQL and MaxDB had multiple repositories, each was branch for a major release [75]. Each branch includes all of the revisions of the previous branch up til the fork point.

Sometimes revisions (bug fixes) from later branches are back ported. We attempted to handle this by independently evaluating each branch and by having a majority voting table.

Revision classes such as documentation and build often do not have many release time revisions. This means that small changes in behaviour or results can heavily bias some of our results if we do not have enough revisions to be statistically significant. How significant were our results for major releases when the number of major releases per project was so low?

We looked at multiple interval lengths in 4.4 and summarized them in 4.5. Our results could have suffered from bias in interval length choice as well as the fact that larger intervals contain all the information of the smaller interval.

We are comparing projects to each other based on measurements, comparisons, and aggregations of those results which are internal to a project itself. Perhaps our choice of intervals and what makes a major and minor release compounded with the different development process and release schedule of different projects renders our comparison useless. Perhaps this is truly comparing apples to oranges.

We have shown that we are concerned about validity but our method does enable us to talk about what happened based on actual evidence stored within a project's SCS and compare that with other projects.

4.5 Possible Extension

Possible extensions will entail studying more projects, applying more analysis techniques, evaluating the data from different perspectives, and investigating non-release time patterns.

This work needs to be expanded upon to study various internal events like branching, tagging, merging and external events. We need to be able to apply this analysis not only to events but across an iteration of the project such that we can characterize the various behaviours and possible stages of software development a project undergoes during an iteration.

More projects should be analyzed in order to achieve some kind of statistical significance so we can generalize our results about OSS release patterns. We will probably need at least 40 projects before we can generalize about FLOSS processes and release patterns. Other perspectives from which we could evaluate include: the distributions, the authors, the files, co-changes, forks, branches and other events.

We need to do some frequency analysis to see if there are any periodic events which would affect linear regressions. Other work includes creating a deployable tool that extracts, analyzes, and produces reports about a project, and then to see if these results are useful to developers.

4.6 Conclusions

Through the application of our methodology on four FLOSS RDBMSs, and the partitioning of revisions into four classes (source, test, build, and documentation) we observed that release patterns exist within projects, although these patterns are not necessarily observed across projects.

One of the observed release patterns is that the frequency of source revisions generally decreases across a release. This might indicate that at release time, developers divert their file updating efforts to activities such as packaging and distribution, which are not recorded in the SCS repository

The fact that release behaviour differs from project to project suggests that the projects we studied follow different processes and have different properties.

Chapter 5

Learning the Reasons behind Changes

We have argued that we can extract purposes and software development processes from revisions in source control systems. This chapter focuses on what those revisions and commits are. In particular we investigate the purpose behind large commits. The purpose of change provide us with another perspective on the underlying software development behaviours and processes that exist within these repositories. Different kinds of changes are being made for different kinds of purposes.

Our investigation of commits shows that these commits are made for a variety of purposes. The purpose of a commit ranges from implementation changes to maintenance changes, which include adaptive, perfective, and corrective changes. Thus we are able to demonstrate that a mixture of maintenance and implementation changes occur within these repositories. That is, there is concurrent and parallel effort across multiple disciplines within a project. We show that source control system commits can contain enough information necessary to glean the purpose of the change itself.

This chapter also deals with the issue that mining of software repositories research has frequently ignored commits that include a large number of files that we call these large commits. This chapter is about understanding the rationale behind large commits, and if there is anything we can learn from them. To address this goal we performed a case study that included the manual classification of large commits of nine open source projects. The contributions include a taxonomy of large commits, which are grouped according to their intention. We contrast large commits against small commits and show that large commits are more perfective while small commits are more corrective. These large commits provide us with a window on the development practices of maintenance teams [1].

[1]Parts of this research was published in a 2008 Mining Software Repositories paper, "What do large

5.1 What About Large Commits?

What do large commits tell us? Often when studying source control repositories large commits are ignored as outliers, ignored in favour of looking at the more common smaller changes. This is perhaps because small changes are likely to be well defined tasks performed on the software system and perhaps it is easier to understand the intentions behind these changes, and draw conclusions from them.

Yet, large commits happen. If we ignore them in our studies, what are we missing? What information do they contain that can help us understand the evolution of a software project? There is plenty of anecdotal information about large commits. A common cited cause for them is a massive copyright licensing change, or reformatting source code [47].

In this chapter we analyze nine popular open source projects. We study two thousand of their largest commits in an attempt to answer two fundamental questions: what prompts a large commit, and what can we learn from such large commits.

Before this study, when asked what are large commits, we simply answered anecdotally, but now we can provide publicly available examples of large commits. In this chapter, we provide a ranking of the frequency of the classes of commits we found by investigating many large commits from multiple FLOSS projects. This allows future researchers and users to rely on both their own experience and the evidence provided in this chapter rather than just their own anecdotal evidence. We also describe the difference in purpose behind large commits versus small changes.

5.1.1 Previous Work

Swanson [149] proposed a classification of maintenance activities as corrective, adaptive and perfective, which we employ in this chapter. Others [131] have characterized small changes and what they mean in the context Swanson's classification of maintenance, faults and lines of code (LOC).

The work exists within the context of the Mining Software Repositories community. Many of these studies extract information from source control systems such as CVS [46, 162].

Many studies have investigated specific projects in detail and have done in depth case studies of the evolution of specific FLOSS projects [45, 51]. Others have gone about quantifying and measuring change [121, 115, 48] in source control systems (SCS). Many software evolution researchers such as Capiluppi et al [24] and Mockus et al. [121] have studied multiple OSS projects from the perspective of software evolution.

---

commits tell us?: a taxonomical study of large commits" by Abram Hindle, Daniel M. German and Richard C. Holt [74].

Software Project	Description
Boost	A comprehensive C++ library.
Egroupware	A PHP office productivity CMS project that integrates various external PHP applications into one coherent unit.
Enlightenment	A Window Manager and a desktop environment for X11.
Evolution	An email client similar to Microsoft Outlook.
Firebird	Relational DBMS gifted to the FLOSS community by Borland.
MySQL (v5.0)	A popular relational DBMS (MySQL uses a different source control repository for each of its versions).
PostgreSQL	Another popular relational DBMS
Samba	Provides Windows network file-system and printer support for Unix.
Spring Framework	A Java based enterprise framework much like enterprise Java beans.

Table 5.1: Software projects used in this study.

5.2 Methodology

We have two primary research questions:

1. What are the different types of large commits that occur in the development of a software product?

2. What do large commits tell us about the evolution of a software product?

We used case studies as the basis for our methodology. We selected nine open source projects (listed in Table 5.1). These projects were selected based upon three main constraints: a) that they were at least 5 years old, mature projects, with a large user base; b) that their source control history was available; and c) that they represented a large spectrum of software projects: different application domains (command line applications, GUI-based, server), programming languages (PHP, C++, C, Java), development styles (company sponsored–MySQL, evolution–or community development–PostgreSQL).

The first stage (of two stages) of our research consisted of the creation of a classification of commits. We proceeded as follows:

1. For each project, we retrieved their commit history. We then selected the 1% commits, per each project, that contained the largest number of files (of any file type, not only source code) for our manual inspection. We inspected and annotated 2000 commits.

Categories of Change	**Issues addressed.**
Corrective	Processing failure Performance failure Implementation failure
Adaptive	Change in data environment Change in processing environment
Perfective	Processing inefficiency Performance enhancement Maintainability
Feature Addition	New requirements
Non functional	Legal Source Control System management Code clean-up

Table 5.2: Our categorization of maintenance tasks; it extends Swanson's categorization [149] to include feature addition and non-functional changes as separate categories. We refer to this as the Extended Swanson Categories of Changes.

2. We created a classification of large commits. We used Swanson's Maintenance Classification [149] as the starting point. As its name implies, Swanson's Maintenance Classification is mostly oriented towards activities that adapt an existent system. We added two more categories to it: *Implementation* (adding features), and *Non-Functional*. Non-functional refers to changes to the software that are not expected to alter its functionality in anyway, but need to be performed as part of the typical software development cycle. For example, adding comments or reformatting source code. These are summarized in Table 5.2.

3. Based upon these categories of changes, and the issues that they address we proceeded to create a candidate list of types of commits we would expect to find. These can be seen as "low level" descriptors of developer's intentions such as "add feature", "bug fix", "change of license", "reindentation". We refined this list by *manually* classifying the large commits of the first two projects we studied (MySQL and Boost). This classification helped us improve and refine our list of types, which is detailed in Table 5.3. We also discovered that a commit can be of one or more types (frequently a commit contains several independent changes). We then mapped these types of commits back into our Extended Swanson Classification, as shown in Table 5.4.

The Extended Swanson Classification appeared insufficient to classify large commits. We noticed that many of the large commits such as build (bld), module management (add, rmod, mmod), were difficult to place into these categories. In many cases the intention

Type commit	**Abbrev.**	**Description**
Branch	brch	If the change is primarily to do with branching or working off the main development trunk of the source control system.
Bug fix	bug	One or more bug fixes.
Build	bui	If the focus of the change is on the build or configuration system files (such as Makefiles).
Clean up	cln	Cleaning up the source code or related files. This includes activities such as removing non-used functions.
Legal	lic	A change of license, copyright or authorship.
Cross	cross	A cross cutting concern is addressed (like logging).
Data	data	A change to data files required by the software (different than a change to documentation).
Debug	dbg	A commit that adds debugging code.
Documentation	doc	A change to the system's documentation.
External	ext	Code that was submitted to the project by developers who are not part of the core team of the project.
Feature Add	fea	An addition/implementation of a new feature.
Indentation	ind	Re-indenting or reformatting of the source code.
Initialization	init	A module being initialized or imported (usually one of the first commits to the project).
Internationalization	int	A change related to its support for languages other-than-English.
Source Control	scs	A change that is the result of the way the source controls system works, or the features provided to its users (for example, tagging a snapshot).
Maintenance	mntn	A commit that performs activities common during maintenance cycle (different from bug fixes, yet, not quite as radical as new features).
Merge	mrg	Code merged from a branch into the main trunk of the source control system; it might also be the result of a large number of different and non-necessarily related changes committed simultaneously to the source control system.
Module Add	add	If a module (directory) or files have been added to a project.
Module Move	mmod	When a module or files are moved or renamed.
Module Remove	rmod	Deletion of module or files.
Platform Specific	plat	A change needed for a specific platform (such as different hardware or operating system).
Refactoring	rfact	Refactoring of portions of the source code.
Rename	ren	One or more files are renamed, but remain in the same module (directory).
Testing	tst	A change related to the files required for testing or benchmarking.
Token Replace	trpl	An token (such as an identifier) is renamed across many files (e.g. change the name or a function).
Versioning	ver	A change in version labels of the software (such as replacing "2.0" with "2.1").

Table 5.3: **Types of Commits** used to annotate commits. These types attempt to capture the focus of the commit rather than every detail about it; for instance *Token Replacement* that affected 1 build file and 10 source files will not be labelled *Build*. A commit could be labelled with one or more types. Their abbreviation is used in the figures in this chapter.

Category of Change	**Types of Change.**
Corrective	bug, dbg
Adaptive	plat, bld, test,doc, data, intl
Perfective	cln, ind, mntn, mmod, rfact, rmod
Implementation	init, add, fea, ext, int
Non functional	lic, rmmod, ren, trpl, mrg
Other	cross, brch

Table 5.4: An attempt at classifying types of changes according to the Extended Swanson Categories of Change. Some of the types of change do not fit only one category; for example, a documentation change might be adaptive or perfective. See Table 5.3 for a legend of types.

of the commit was none of these. For this reason we decided to create a new taxonomy, which we call *Categories of Large Commits*, which is summarized in Table 5.5. We used this taxonomy to organize the types of commits, as depicted in Table 5.6.

Manually classifying commits is difficult. We are not contributors to any of these projects, and we relied on our experience as software developers to do so. The procedure we used was the following:

- We read the commit log. Most of the commits in these projects have log comments which provide a good rationale behind the commit. In some cases the log was too explicit (for example, the largest log comment in Evolution was 13,500 characters long), but in many cases the commit was simple and clear in its intention (e.g. "HEAD sync. Surprisingly painless","Update the licensing information to require version 2 of the GPL").
- We identified the files changed. Usually the filename (and its extension) provides certain clues (e.g. *.jpg are most likely documentation or data files).
- We studied the `diff` of the commit, and compared its contents to the commit log. We believe that the contrasting of both sources of information improved the quality of our classification. If they appeared to contradict each other we used the information in the diff.

The result of this classification was, for each commit, a list of types of commits that reflected the intention of a commit.

For the second stage of our study we proceeded as follows:

Categories of Large Commits	**Description**
Implementation	New requirements
Maintenance	Maintenance activities.
Module Management	Changes related to the way the files are named and organized into modules.
Legal	Any change related to the license or authorship of the system.
Non-functional source-code changes	Changes to the source code that did not affect the functionality of the software, such as reformatting the code, removal of white-space, token renaming, classic refactoring, code cleanup (such as removing or adding block delimiters without affecting the functionality)
SCS Management	Changes required as a result of the manner the Source Control System is used by the software project, such as branching, importing, tagging, etc.
Meta-Program	Changes performed to files required by the software, but which are not source code, such as data files, documentation, and Makefiles.

Table 5.5: Categories of Large Commits. They reflect better the types of large commits we observed than those by Swanson.

Categories of Large Commits	**Types of Commits**
Implementation	fea, int, plat
Maintenance	bug, dbg, mntn, cross
Module Management	add, mmod, rmod, split
Legal	lic
Non-functional source changes	cln, trpl, rfct, indent
SCS Management	brnch, ext, merge, ver, scs
Meta-Program	bui, tst, doc, data, intl

Table 5.6: Classification of the types of commits using the Categories of Large Commits

1. For each project we retrieved (with the exception of MySQL[2]) the log for each commit, and the corresponding diff to its files.

2. We proceeded to *manually* classify each of these commits into one or more of the different types of large commits (as shown in Table 5.3). Every commit was labelled with one or more types.

3. For each project we created a summary of what we consider the "theme" of the large commits. In other words, we tried to draw a rationale that explained why the project was doing large commits, and what such commits explained about the project, its organization, development process, or its developers (qualitative analysis).

4. We quantitatively analyzed the resulting data, both by project, and as an average over all projects.

5.3 Results

We present the results of our study in two parts. In the first part we describe the themes of our qualitative results. In the second part we present an statistical overview of the types of large commits.

5.3.1 Themes of the Large Commits

By reading and classifying the log-changes we were able to draw certain conclusions of the rationale behind many of the large commits. We refer to these conclusions as the *themes* of the large commits of each project. Each project had its own themes:

[2]MySQL uses `Bitkeeper` as source control system, and we were not able to retrieve the diffs for each commit.

Boost's documentation produced many large commits because they were auto-generated from the source code and docbook files (manuals). Boost developers used branches and consequently many of the large commits were merges from branches to the trunk. A common yearly large commits was an update to the year of the copyright. Boost also changed its license partway through development. Boost has a well-defined source code style and we observed many cleanup commits (reformatting and rewrites to follow their style). Refactorings and unit tests were also common large commits.

MySQL uses one repository per version. Its source code is a "fork". (Version 5.0 was forked from Version 4.1). Its use of Bitkeeper allowed a type of commit that we did not observed in any other project: changes in file permissions. MySQL programmers used a consistent lexicon and marked bug fixes with the word "fix" and merges with the word "merge". Merges were a very common type of large commit.

Firebird had many large build commits because of their support for multiple platforms. Their use of Microsoft Visual C++ project files often created as many build files as there were source files. These project files were also very prone to updating. Firebird had many large commits which were both module additions and cross cutting changes. Modules which updated or provided memory management, logging or various performance improvements were added and caused large cross cutting, far reaching changes. Since Firebird was inherited from Borland, there were many large commits where old code was cleaned-up to follow the new coding style. A couple of large commits included the dropping of some legacy platform support.

Samba stores most of its documentation in the SCS and many of the documents were auto-generated from other documents; as a consequence many of the large commits were documentation driven. Samba had multiple branches developing in parallel, often new functionality was back-ported from development branches to legacy branches. Thus merging was a common large commit for Samba. Large build and configuration commits included support for Debian builds, requiring a separate build module. Samba also went through a few security audits and the addition of cross cutting features such as secure string functions and alternative memory management routines. Except for the Debian related commits, we were surprised to see very few platform support commits as Samba runs on almost any flavour of Unix.

Egroupware integrates various external PHP applications into one coherent unit. Egroupware consistently had a large number of template changes (in many cases the templates contained code). Many large commits were due to the importing of externally developed modules which would had to be stylized and integrated into Egroupware. The integration of external modules also implied a lot of merging of updated external modules. There was also a surprising number of commits where only version strings were changed.

Enlightenment is a window manager that is known for fancy graphics and themed window decorations. "Visual" themes (a group of images, animations and source code defining a look) are the most notable part of the large Enlightenment commits. Large window manager themes are imported and updated regularly. Enlightenment uses separate directories to do versioning and merges are often copying or moving files from one directory to another. Enlightenment also attracted a lot of small widgets and tools that shared Enlightenment visual themes, like clocks and terminal emulators. Some large commits consisted of imports of externally developed tools into the repository. There were very few tests in Enlightenment.

Spring Framework had many large commits that consisted of examples and documentation. An example application, Pet Store, was added and removed multiple times. Branching in the Spring Framework often was done at the file level, with many large commits being moved from the sandbox directory to the main trunk. Much Spring Framework code was associated with XML files that configured and linked spring components. Other very common large commits were refactorings, renamings, and module and file moves. Spring required large cross cutting changes.

PostgreSQL was distinct for its emphasis in cleaning up its source code. Many of its large commits were code reformatting and code cleanup. Such changes included removing braces around one statement blocks as PostgreSQL is written in C. Many of its large commits involved a single feature (suggesting cross-cutting concerns). Many of these commits were external contributions (from non-core developers).

Evolution makes extensive use of branching: many of its large commits reflect the use of branches for different versions, and to develop and test new features before they are integrated into the main trunk of the source control system. It also uses many data files (such as time-zone information) that are usually updated at the same time. Evolution uses clone-by-owning: it makes a copy of the sub-project *libical* every time this project releases a new version.

General themes that we could extract from these projects were that many changes relate to: improving readability, merging externally produced projects and libraries, testing, clone by owning, licensing changes, build system changes, and file-based modularity of languages such as Java.

5.3.2 Quantitative Analysis

In this section we analyze the quantifiable properties of the categorizations of the changes that we extracted from these languages.

Types of Changes

We first present the distribution for each of the types of changes. Figure 5.1 shows their proportion by project. Each bar corresponds to all changes for that type, and it is divided per project. For example, for maintenance changes, 60% are to Spring Framework, and the rest 40% to Evolution. Figure 5.2 groups them together, as a percentage of the total number of commits for all projects. Maintenance changes are slightly less than one percent of all commits. They show a lot of variation, but the most frequent changes were Adding Modules, Documentation, Initializing Modules, Feature Additions, and Merges. Table 5.3 shows the corresponding percentage of commits for each type in descending order. When comparing both figures, it is interesting to see that those types of changes dominated by one or two applications (such as Data, Split Module, Debug, Maintenance) occur very rarely.

Extended Swanson Maintenance Categorization

Figure 5.3 shows, for each project, the distribution of changes according to the Extended Swanson Maintenance Categories (see Table 5.2). Figure 5.4 shows the aggregation of all projects. All projects show commits in each category, although in some cases (such as Spring and Firebird) they were dominated by one or two categories.

Table 5.8 shows what percentage of large commits belonged to each Extended Swanson category. We compared our results (ignoring non-functional and implementation changes) to Purushothaman et al.'s[131]. Purushothaman divided changes into: Corrective, Perfective, Adaptive, Inspections, and Unknown. They found that 33%, 8%, 47%, 9%, 2% of small changes fell into each of these categories. We found the following: adaptive changes were the most frequent in both large and small changes; Small changes, however, were more likely to be corrective than perfective. For large commits the trend is inverted: perfective changes are more likely to occur than corrective changes. This result seems intuitive: correcting errors is often a surgical, small change; while perfective changes are larger in scope and likely to touch several files.

Categorization of Large Commits

Figure 5.5 shows the distribution of changes when they were classified according to our categories of large commits (see Table 5.5). Projects vary by the distributions of the different categories. These distributions appear to support qualitative findings described in Section 5.3.1; for example, Boost has the largest proportion of Meta-Program (mostly documentation) while PostgreSQL has the largest proportion of Non-functional Code. This is likely due to its frequent reformatting of the code.

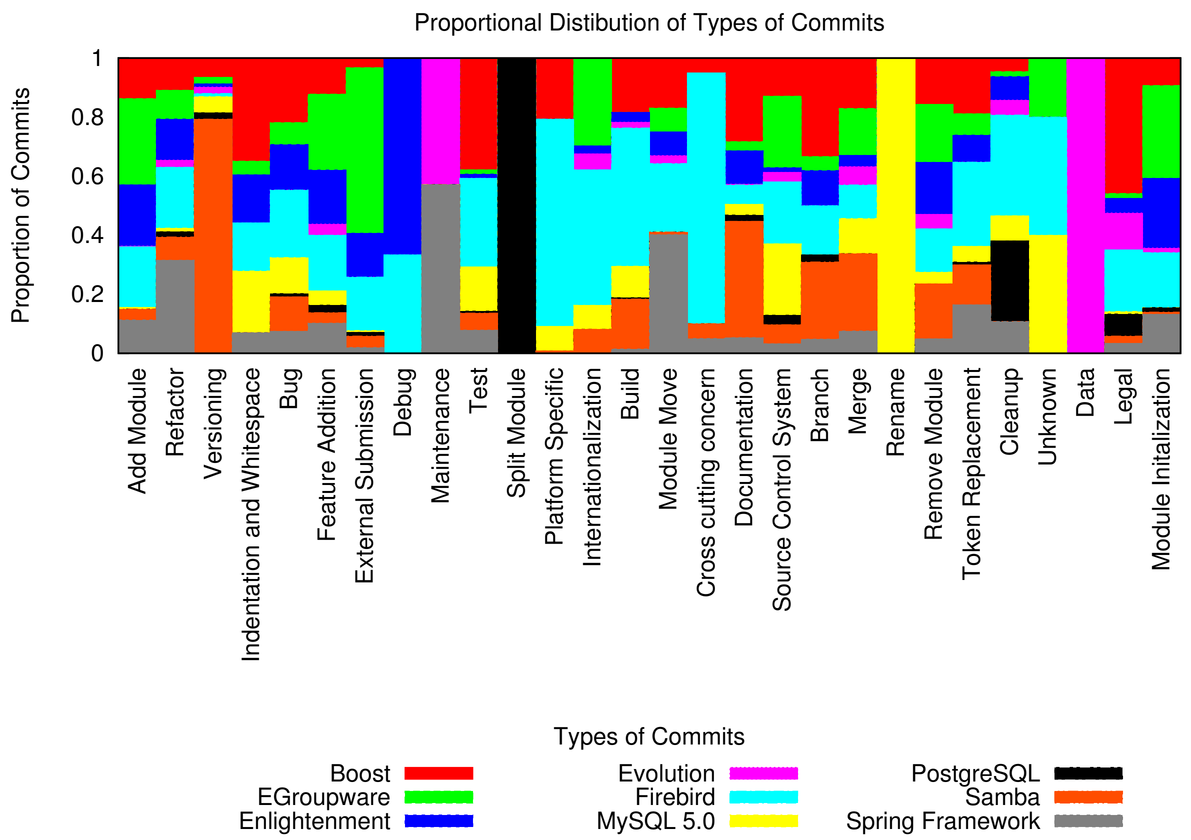


Figure 5.1: Distribution of Types of Changes by project. Each bar corresponds to 100% of commits of that type and it is divided proportionally according to their frequency in each of the projects.

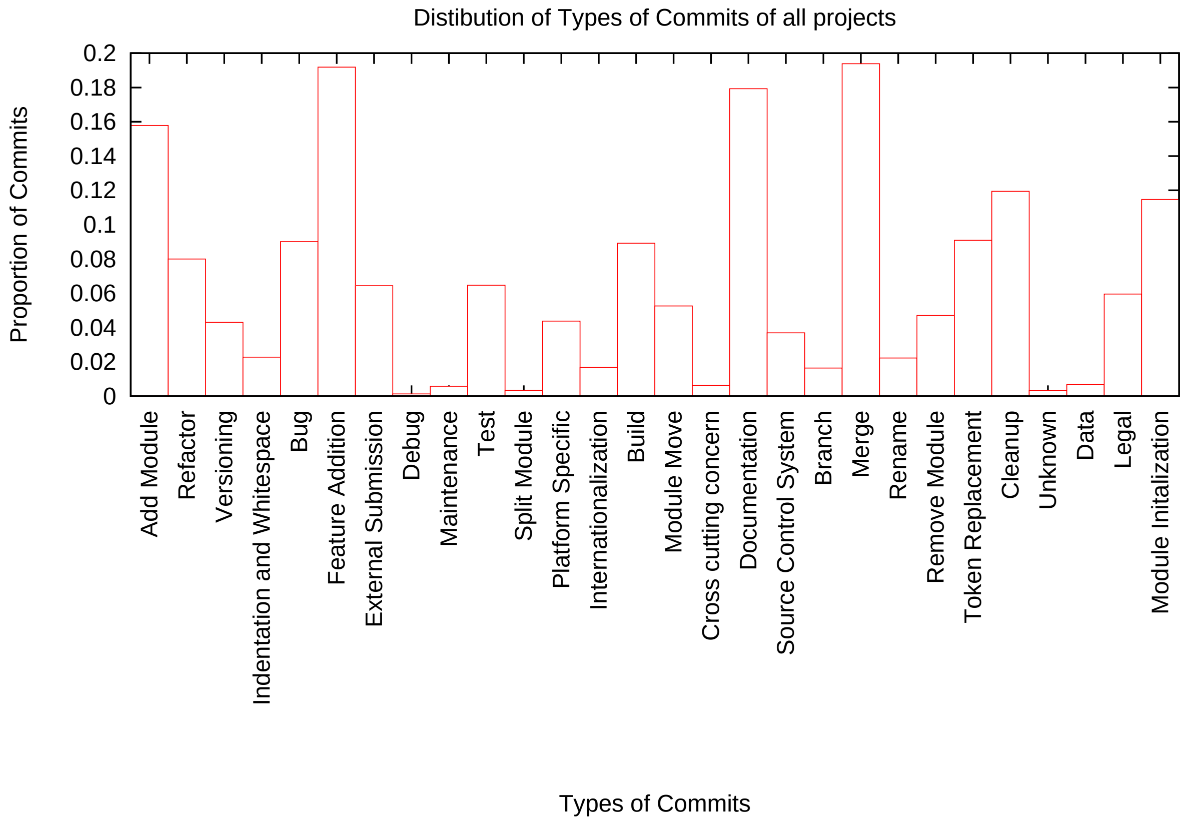


Figure 5.2: Distribution of Types of Change of the aggregated sum of all projects.

Type of Change	Percent
Merge	19.4 %
Feature Addition	19.2 %
Documentation	17.9 %
Add Module	15.8 %
Cleanup	11.9 %
Module Initalization	11.5 %
Token Replacement	9.1 %
Bug	9.0 %
Build	8.9 %
Refactor	8.0 %
Test	6.5 %
External Submission	6.4 %
Legal	6.0 %
Module Move	5.3 %
Remove Module	4.7 %
Platform Specific	4.4 %
Versioning	4.3 %
Source Control System	3.7 %
Indentation and Whitespace	2.3 %
Rename	2.2 %
Internationalization	1.7 %
Branch	1.6 %
Data	0.7 %
Cross cutting concern	0.6 %
Maintenance	0.6 %
Split Module	0.3 %
Unknown	0.3 %
Debug	0.1 %

Table 5.7: Distribution of commits belonging to each Type of Change over all projects (1 commit might have multiple types).

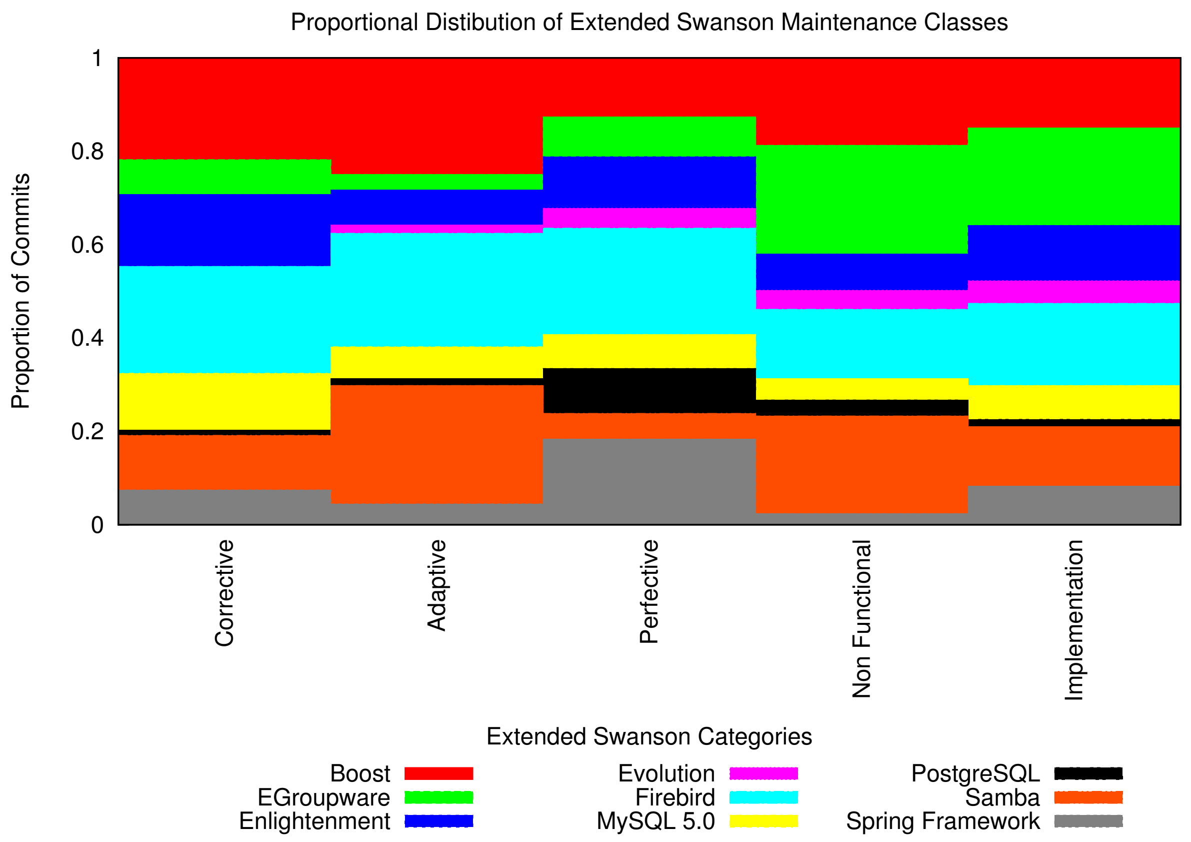


Figure 5.3: Distribution of changes per project, organized using Extended Swanson Maintenance Categories.

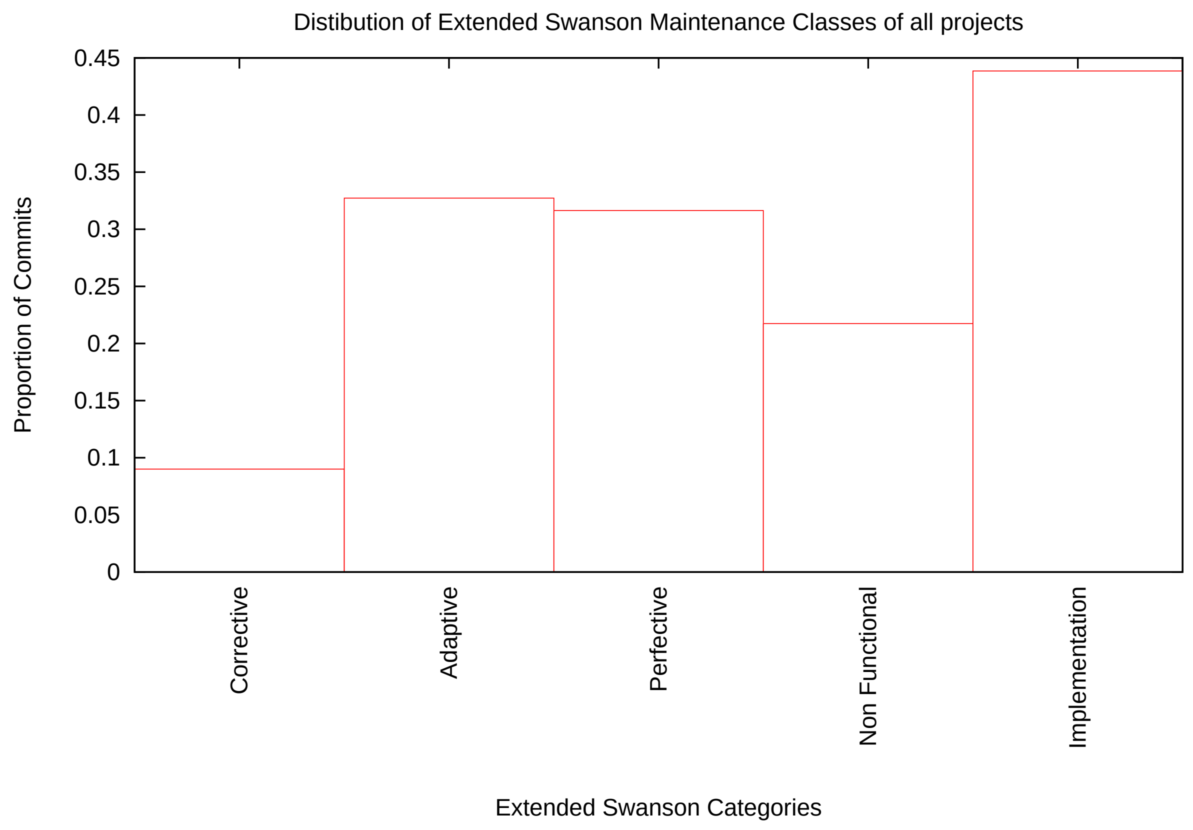


Figure 5.4: Distribution of changes for all projects, organized using Extended Swanson Maintenance Categories

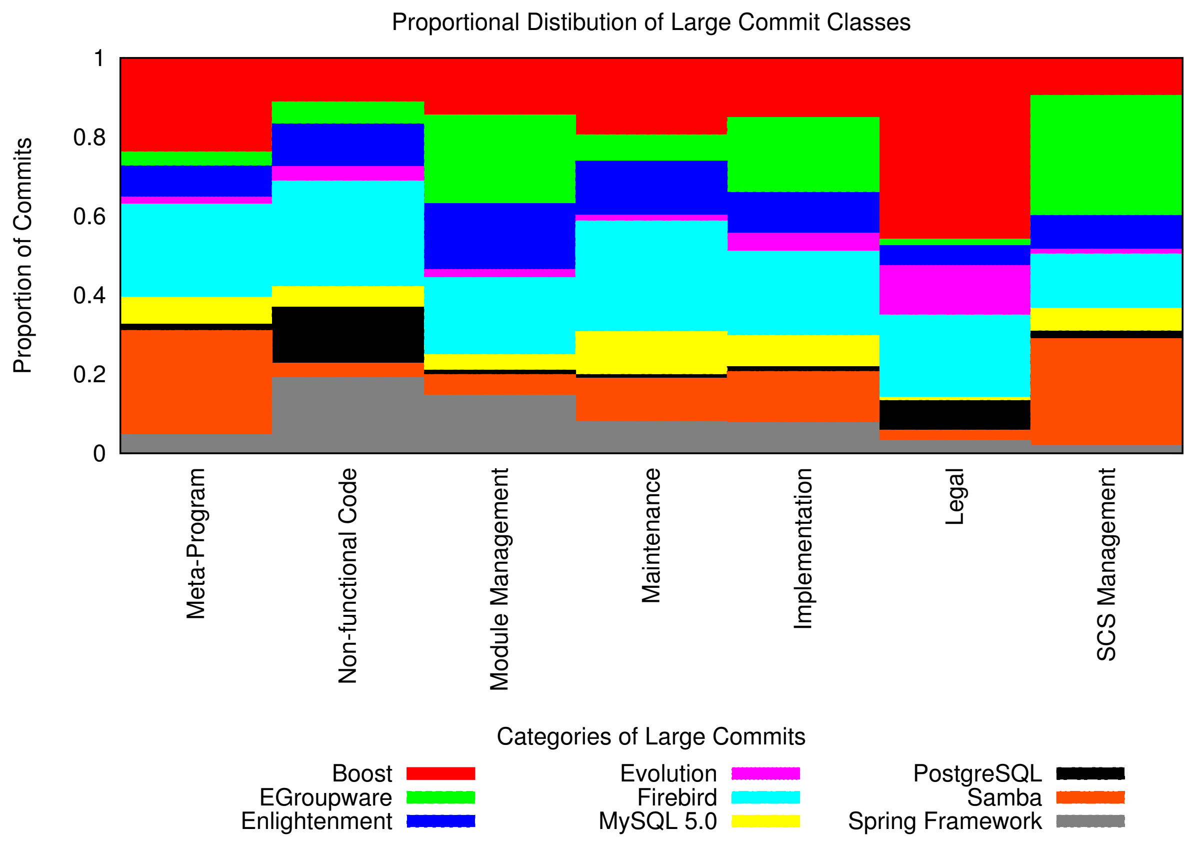


Figure 5.5: Distribution of commits per project, classified according to the Categories of Large Commits.

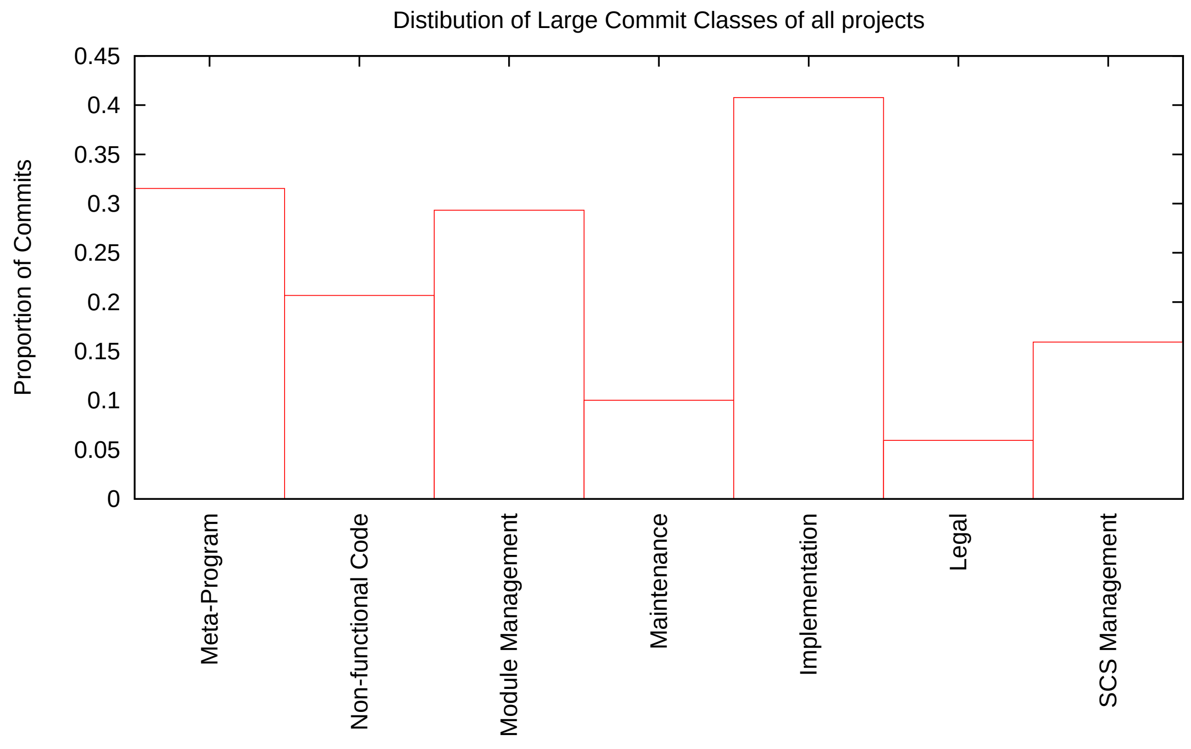


Figure 5.6: Distribution of commits for all projects, classified according to the Categories of Large Commits.

Category	Percent
Implementation	43.9 %
Adaptive	32.7 %
Perfective	31.6 %
Non Functional	21.7 %
Corrective	9.0 %

Table 5.8: Proportion of changes belonging to each Extended Swanson Maintenance Category for all projects (1 commit might belong to multiple ones).

Category	Percent
Implementation	40.8 %
Meta-Program	31.5 %
Module Management	29.3 %
Non-functional Code	20.7 %
SCS Management	15.9 %
Maintenance	10.0 %
Legal	6.0 %

Table 5.9: The percent of commits belonging to each Large Maintenance Category over all projects (1 commit might belong to multiple categories).

Figure 5.6 shows the accumulated distribution of all the projects (summarized in Table 5.9). The most common category is Implementation, followed closely by Meta-Program and Module Management. While some of the largest commits might be Legal, they represent the smallest category.

5.4 Analysis and Discussion

Large commits occur for many different reasons, and these reasons vary from project to project. They tend to reflect several different aspects:

Their development practices. Branching and merging results in large commits, but not all projects use it. Branching and merging is primarily used to provide a separate area for development, where a contributor can work without affecting others. Once the developer (and the rest of the team) is convinced that the code in the branch is functioning properly (e.g. the feature or features are completed) the branch is merged back to the trunk (the main development area of the repository). If a project uses branching and merging is because (we hypothesize) they prefer to develop and test a feature independently of the rest of the project, and then commit to the trunk a solid, well debugged large commit–

instead of many, smaller ones. Branches become sandboxes. Evolution is an example of such project.

Their use (or lack of use) of the source control system. Many projects extensively use the features of a source control system, and this is reflected in the large commits (maintaining different branches for each version of the product–such as Evolution), while others do not (Enlightenment maintains different directories for different versions). Some projects use different repositories for each version (e.g. MySQL).

The importance of code readability. Some projects worry a lot about the way their code looks (Boost and PostgreSQL). These project have regular commits that make sure the code obeys their coding standards.

Externally produced features. Some projects accept a significant number of contributions from outsiders (to the core team) and these are usually committed in one single large commit (e.g. PostgreSQL, Egroupware).

Automatically generated files. Many large commits are the result of automatically generated files (Boost and Samba). This could be a concern for researchers creating tools that automatically analyze software repositories without properly determining the provenance of a given file.

Test cases and examples tend to be added in large commits. It is also interesting that not all projects have test suites nor examples where appropriate.

Clone-by-owning results in regular large commits. Some projects keep a local copy of another product that they depend upon, such as a library, but they do not maintain it; such products need to be regularly updated with newer versions[3]. These updates are performed on a regular basis, and result in large commits. A project can look to the outsider as if it is having significant activity, when in reality is just copying code from somewhere else.

The copyright and licensing changes can be useful in tracking the legal provenance of a product. For example, who have been their copyright owners, or its changes in license. For example, Evolution has had many different copyright owners during its lifetime, while Boost changed its license.

The development toolkit has an impact on large commits. We noticed that the use of Visual Studio prompted significantly more large commits to build files, compared to cmake or autoconf/automake (used by many projects).

The impact of the language. Large commits for languages such as C++ and Java contained a lot of refactorings and token replacement. This might be not a result of the

[3]Clone-by-owning is usually done to avoid unexpected changes to the original software; it can also be used to simplify the building process.

feature of the language, but the manner in which programmers of that language tend to work. Perhaps refactoring is more likely to be performed by programmers of object oriented languages; it is also possible that C++ and Java programmers were more likely to use the term “refactoring” in their commit logs, and this had an effect in the way we classified their commits. Spring, Firebird and Boost, written in Java and C++, contained many API changes and refactorings.

We believe that reading a commit log and its diff gives an idea of how easy or difficult it is to maintain a system. For example, we observed that several features in PostgreSQL required large commits to be implemented. This is very subjective, but reliable methods could be researched and developed to quantify such effect.

5.4.1 Threats to Validity

Our study unfortunately suffers many threats to validity. Our main threats were were consistency in annotation, subjectivity of annotation, and miscategorization. Much of the annotation relied on our judgement, as programmers, of what the commit probably meant. In the case of MySQL we did not use the source code, just the filenames and change comments. We annotated to the commits with categories and in many cases these categories could be assigned subjectively. There were also only two people, Hindle and German, annotating the data, thus there might be bias and disparity in the annotation step.

Another issue is our choice of projects, we only chose nine relatively large FLOSS projects. Do our results scale up and down? Do our results apply to non-open source products?

5.5 Conclusions

Although large commits might look like outliers in the large data-sets extracted from SCSs we have shown that in many cases these commits are fundamentally important to the software’s very structure. Many of the large commits actually modify the architecture of the system.

We compared our study with another study of small changes and found an important difference between small changes and large commits. The difference was that large commits were more likely to perfective than corrective, while small changes were more often corrective rather than perfective. In a way it makes sense, correcting errors is surgical, perfecting a system is much more global in scope.

We have shown that the large commits provide insight into the manner in which projects are developed and reflect upon the software development practices of its authors.

The next step is to leverage this data-set to automatically classify changes by their maintenance categories. In the next chapter we exploit this data-set further. We apply a lexical analysis of the diffs and change comments, in an attempt to automatically classify large commits.

Chapter 6

Classifying Changes by Rationale

This chapter helps demonstrate how we can automatically recover the purpose and behaviour behind many changes. This chapter is much like Chapter 5 except we automate classification and validate this automation of classification. This chapter relies on the annotations from the previous chapter. We further our argument that process exists, it is observable and different events related to different disciplines such as such as implementation, maintenance, and bug fixing are mixed throughout the life-cycle of many projects.

We show that commits contained within source control systems contain enough information to be automatically classified by their purpose. Their purposes are essentially the behaviour of the developers who are making the changes.

We show that commits can be partitioned and classified into many classes of changes related to maintenance and implementation. Thus we demonstrate that we can extract behaviours from the repository that are related to implementation and maintenance.

Large software systems undergo significant evolution during their lifespan, yet often individual changes are not well documented. In this work, we automatically classify large changes into various categories of maintenance tasks — corrective, adaptive, perfective, feature addition, and non-functional improvement — using machine learning techniques. In the previous chapter, we found that many commits could be classified easily and reliably based solely on the manual analysis of the commit meta-data and commit messages (i.e., without reference to the source code). Our extension is the automation of classification by training machine learners on features extracted from the commit meta-data, such as the word distribution of a commit message, commit author, and modules modified. We validated the results of the learners via 10-fold cross validation, which achieved accuracies consistently above 50%, indicating fair to good results. We found that the identity of the author of a commit provided much information about the maintenance class of a commit, almost as much as the words of the commit message. This implies that for most large commits, the Source Control System (SCS) commit messages plus the commit author identity is enough information to accurately and automatically categorize the nature of the maintenance task [1].

6.1 On the Classification of Large Commits

Large commits are those in which a large number of files, say thirty or more, are modified and submitted to the Source Control System (SCS) at the same time. We demonstrated that large commits provide a valuable window into the software development practices of a software project [74]. For example, their analysis can highlight a variety of identifiable behaviours, including the way in which developers use the source control system, including branching and merging, the incorporation of large amounts of code, such as libraries, into the code, and continual code reformatting.

While any large change might superficially seem to be significant due to its “size”, however defined, it is important to recognize that not all large changes are created equal. For example, a commit that updates the stated year of the copyright for a system within the boilerplate comments might touch a very large number of files, but such a commit has no semantic impact on the system. Similarly, a change that performs a simple renaming of an oft-used identifier is less risky than a smaller but invasive refactoring of the program’s design.

To aide further analysis, it is necessary to identify the type of change that a large commit corresponds to. For example, a developer who is interested in tracking how the code’s design has evolved will probably want to ignore changes to white-space, comments or licensing. This is particularly important for co-change analysis of changes (for a survey of such methods see [88]), where the inclusion of two entities, such as files, methods, or functions, in the same commit is expected to be significant. The relationship between two files that were changed during a change in white-space, perhaps while reformatting of the source code, is probably negligible when compared to the relationship of two files changed during refactoring. Classifying large changes, which we did manually in Chapter 5, is useful to select or to ignore certain large changes during the analysis and exploration of such changes.

A major challenge, however, is how to automatically classify such large commits. The simplest solution would be for developers to manually tag commits with meta-data that indicates the type of commit performed. While this is feasible in an organization that

[1]This work is derived from the paper “Automatic Classification of Large Changes into Maintenance Categories” by Abram Hindle, Daniel M. German, Michael W. Godfrey and Richard C. Holt, published in ICPC 2009 [73]

dictates software development practices, it does not provide a solution to the large corpus of commits already performed.

Another solution is to manually classify previous large commits. In Chapter 5, we manually annotated 2000 large commits. For a given project this may be feasible. Large commits correspond usually to the top 1% of commits with the most files changed. However, a manual approach may not scale well for some systems; consequently, we decided to investigate automatic classification of large changes.

When a programmer commits a change, they may disclose the intention of a change in the commit message. Often these commit messages explain what the programmer did and what the intended purpose of the change was. If it is a large change, one might expect a programmer to document the change well.

We show in this chapter that large source control system (SCS) commit messages often contain enough information to determine the type of change occurring in the commit. This observation seems invariant across different projects that we studied, although to a different extent per each project.

Classifying commits using only their meta-data is attractive because it does not require retrieving and then analyzing the source code of the commit. Retrieving only the meta-data of the commit is significantly less expensive, and allows for efficient browsing and analysis of the commits and their constituent revisions. These techniques would be useful to anyone who needs to quickly categorize or filter out irrelevant revisions.

6.2 Previous Work

This work extends our work on Large Commits classification [74], discussed in Chapter 5, and the classification work of Robles et al. [6]. We rely on the data produced by our Large Commits study to execute this study.

These works derive from Swanson's Maintenance Classification [149], that provided a taxonomy for maintenance changes (see Table 5.2), the study of small changes by Purushothaman et al. [131], and later work by Alali et al. [5]. Purushothaman et al. extended the Swanson Maintenance Classification to include code inspection, then they classified many small changes by hand and summarized the distribution of classified revisions. Alali et al. further studied the properties of commits of all sizes but did not classify the commits. Our previous work [74] classified large commits by their maintenance type; we discovered that many large commits are integration and feature addition commits, which were often ignored as noise in other studies. We made multiple categorization schemes similar to Swanson (see Table 5.5 for the Large Changes categorization). Robles et al. had a similar idea and extended the Swanson categorization hierarchically.

6.3 Methodology

Our methodology can be summarized as follows. We selected a set of large, long-lived open source projects; for each of them we manually classified 1% of their largest commits, the commits with the most files changed. We used these manually classified commits to determine if machine learning was useful to automatically classify them. The details are described in this section.

6.3.1 Projects

The projects that we selected for this study were widely used, mature, large open source projects that spanned different application domains (and were representative of each of them). These projects are summarized in Table 5.1 in the previous chapter. The projects are also implemented in a variety of programming languages: C, C++, Java and PHP. Some projects were company sponsored, such as MySQL and Evolution.

6.3.2 Creating the training set

For each project we extracted their commit history from their corresponding source control repositories (CVS or Bitkeeper). We ranked each commit by number of files changed in a commit and selected the top 1% of these commits. We manually classified about 2000 commits into the Extended Swanson Classification (described in Table 5.2), Large Changes classification (see Table 5.5), and detailed large commits classification. This process is described in detail in [74] and the previous Chapter 5.

We took this annotated data-set and wrote transformations to generate the data-sets we wanted to use for automatic classification. We produced data-sets such as the Word Distribution data-set and the Author, File Types and Modules data-set.

6.3.3 Features used for classification

We used the following features for the data-sets used to train the classifiers. Each feature was extracted for each project.

Word Distribution By word distribution we mean the frequency of words in the commit message. We wanted to know if Bayesian type learners were useful for automatic classification. Bayesian learners are used in email spam filters to discriminate between SPAM (undesirable) and HAM (desirable) type messages. These learners determine the probability of a word distribution of a message (the word counts) occurring in both the HAM and SPAM data-sets. Per each large commit we counted the number of occurrences of each word and the number of files that changed in that commit. We kept only tokens consisting of letters and numbers, and no other stop words were removed.

Author The identity of the commit's author. We suspected that some authors were more likely to create certain types of large commits than others.

Module and File Types Is the kind of commit affected by the module, that is the directory where the commit primarily occurred? These features were counts of files changed per directory, thus given a system each directory would become a feature of file counts for that directory. Along with a count of files changed per directory, we also count the files by their kind of usage: source code, testing, build/configuration management, documentation and other (STBD [78] in Chapter 4). These counts are represented by five respective features that are the counts of changed files for each type.

For each project, we created data-sets from each of these features and their combinations (for example, one data-set was created for the word distribution and authors; another for all the features). We also created a data-set with the words distributions in the commit messages for the union of all commits for all projects. Authors and modules were too specific to each project to be useful when considering all the commits. This would permit us to determine if the automatic classification was applicable to all the projects, or if it worked better for some than for others.

6.3.4 Machine Learning Algorithms

Each of these data-sets was fed into multiple machine learning algorithms. We wanted to know which algorithms performed better across the different projects, or if different algorithms were better for different projects). We used the various machine learners from the Weka Machine Learning framework [84]. We use 10-fold cross validation to train a model on 90% of the data and then test the other 10% against this created model. Thus for each learner it created 10 different models and executed 10 different test runs. In total, we used seven Machine Learners:

- J48 is a popular tree-based machine learner (C4.5). It makes a probabilistic decision tree, that is used to classify entities.
- NaiveBayes is a Bayesian learner similar to those used in email spam classification.

- SMO is a support vector machine. Support Vector Machines increase the dimensionality of data until the data points are differentiable in some dimension.
- KStar is a nearest neighbour algorithm that uses a distance metric like the Mahalanobis distance.
- IBk is a single-nearest-neighbour algorithm, it classifies entities taking the class of the closest associated vectors in the training set via distance metrics.
- JRip is an inference and rules-based learner (RIPPER) that tries to come up with propositional rules which can be used to classify elements.
- ZeroR is a learner used to test the results of the other learners. ZeroR chooses the most common category all the time. ZeroR learners are used to compare the results of the other learners to determine if a learners output is useful or not, especially in the presence of one large dominating category.

6.4 Results

The data-sets were run against each of the machine learners. We used five metrics to evaluate each learner: *% Correct* is what percentage of commits were properly classified, which is both the recall and the accuracy; *% Correct ZeroR* was the accuracy of the ZeroR classifier against the same data-set — it is expected that a learner that is classifying data well should always do better than ZeroR; Δ ZeroR is the difference between the *%Correct* and *%Correct Zero* accuracies; *F-1* is the F-Measure, a value between 0 and 1 produced by a combination of precision (instances were not misclassified) and recall (total correctly classified instances); and *ROC*, which is the area under the Receiver Operating Characteristic (ROC) curve–this value is similar to the F-Measure and it is based on the plotting of true positives versus false positives, yet the end value closely mimics letter grades in the sense of the quality of classification, e.g., 0.7, a *B* grade, is considered fair to good.

Table 6.1 shows the results of the best learner using the data-set *Words distributions* per project. As it can be observed, no single classifier is best; nonetheless, usually the results were better than the ZeroR classifier which indicates that some useful classification was being done by the learner. Table 6.2 shows that the results are less accurate for the *Authors and Modules* data-set, but still better than ZeroR.

The *Word Distribution* data-set appears to be better at producing successful classifiers than the second *Authors and modules* data-set.

6.4.1 Learning from Decision Trees and Rules

The learners provided valuable output about their models and their classifications. Some of them helped by highlighting the most useful information they use to do the classification. In particular, the output of tree- and rules-based learners provides some interesting insights.

The J48 tree learner produced decision trees based on the training data. The trees created for the word distribution data-sets are composed of decisions that depend on the existence of specific words, such as *cleanup* or *initial*. Specific authors and directories were also commonly used by decisions in these trees.

The decision trees output by the J48 learner contain one word in each of its nodes; we tabulated such words and used their frequency to create tag clouds or text clouds. Figure 6.1 depicts the text cloud of words used in the union of all projects, while Figure 6.2 corresponds to the text clouds for each project. Words related to implementation changes were the most frequent, such as "add", "added", "new". Other frequent words relate to *refactoring*, *bug fixing*, *documentation*, *cleanup* and *configuration management* changes.

We believe this suggests that simple word match classifiers can automate the previously manual task of classification.

The JRip classifier is a rules-based classifier that infers a small set of rules used to classify instances. With the word-based data-set, JRip produced rule-sets which contained both expected rules and sometimes unexpected rules. The unexpected rules were often the most interesting.

For the large-changes classification, if the commit message contained words such as "license" or "copyright", the commit message would be classified as a *Legal* change. If the commit message contained words such as (or related to) "fixes", "docs" and "merging" it was classified as *Maintenance*. If the commit message contained words such as "initial" or "head" it was classified as *Feature Addition*. "Translations", "cleanup", "cleaning", "polishing" and "whitespace" were words associated with *Non-functional* changes.

One interesting result was that for PostgreSQL, the classifier properly classified commit messages that contained the word "pgindent" (the name of its code indenter) as *Non-Functional*.

For the data-set of *Authors, File Types and Directories*, we found that author features were most frequently part of rules. For instance in Firebird "robocop" was associated with *Non-functional* changes and "skywalker" was associated with *Module-Management*, in Boost, "hkaiser" was associated with *SCS-Management*. The counts of file types such as "Source", "Test", "Build", "Documentation" and "Other" we used in some decision rules. Often changes with low counts of "Other" files were believed to be *Non-functional* changes, but those with many "Other" files were considered *Module-Management* changes.

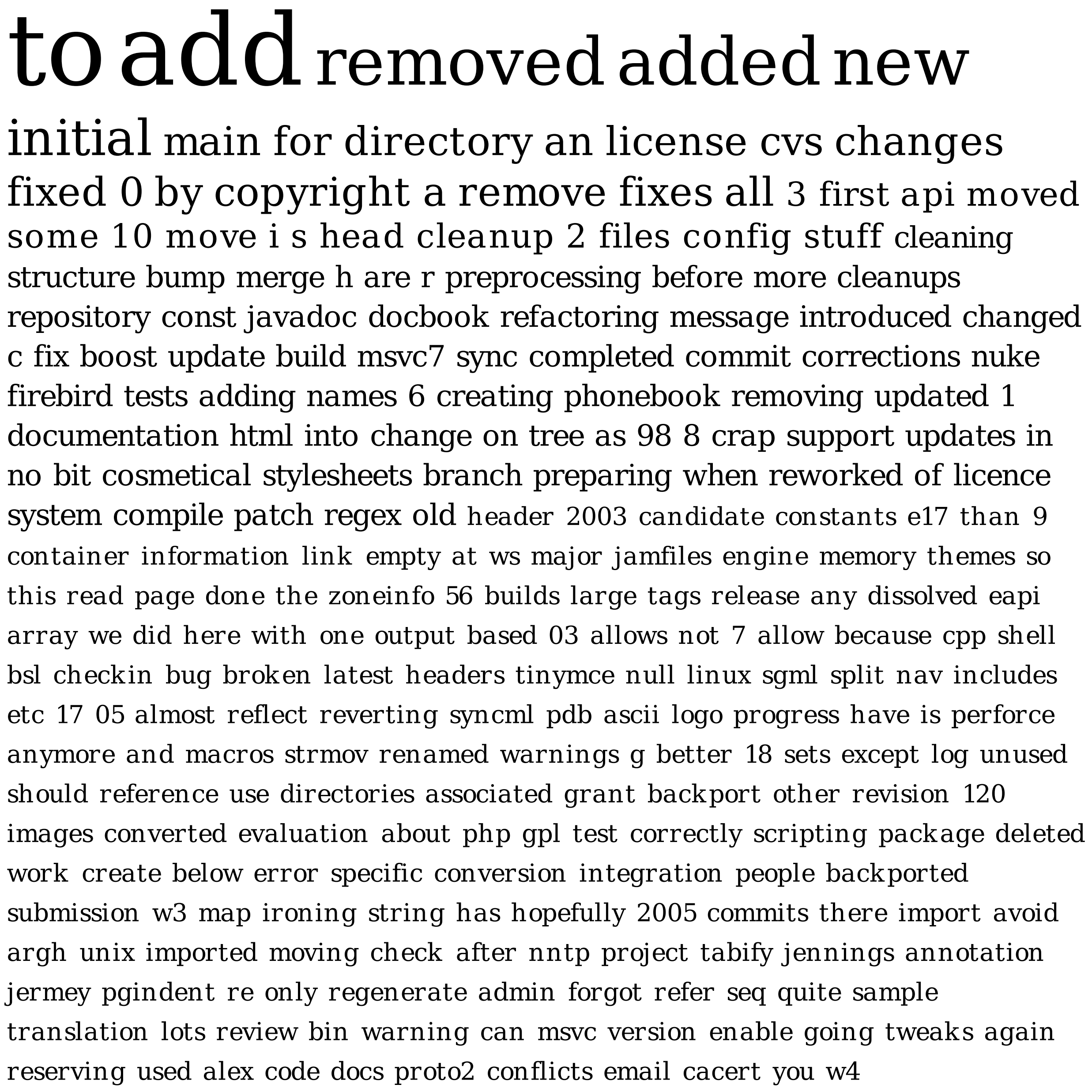


Figure 6.1: Text Cloud of tokens used by the J48 learner to classify commits. The size of the text indicates how many instances of the word occurred in trees produced for the union of all the projects' commits.

move config license merge removed fixes proto2 refactoring cvs 0 checkin updates 1 2 copyright almost cosmetical merged to perforce specific fixed licence and cpp 4 9 3 link tabs borland creating revision by preprocessing head converted proposal copyrights directory review ascii v1 part progress sync documentation into 295 regex commit change line docs submission gif initial tests candidate phoenix2 message bump hpp unix contest friendly 6 empty moving problem apply library pre add dunno headers support graph jamfiles class building separate main default for removing links cleanup seq doc updated minmax bbc2 accessor tr2 miniboost boost a remove update reverting style tweaks reworked linefeeds from good is all fusion attributes again reserving dots name renamed also cr on

(a) Boost

head added fixed as plugin crap dev4 for new directory update removed 12 old better apps around 4 51 first 6 moving 0 been after add additional sitemgr switching default identical converted incl updated catalan php remove the now index cleanups plus 52 1240675 branch fixes 50 places set 65 documentation to phpgwapi dir svn of anymore system and reverted labbe based amazon docs addressbook applying rc1 hopefully

(b) EGroupware

api stuff updates a nav page all indent warnings nuke 2005 structure e17 argh add ie split cleanup quite removed fixes html we fixed allow breaks as adding there strings bump first direcotry merging ability mon 0 dead merge doing keys bug works headers 22x22 data setting for may 28 correct head themes images forgot imenu this edb about copyrights lately seb 17 foucs icons actually copyright bit test update ones cleanups asm work eapi 2000 have anyway menus wed do added pages download autof00 browsers patch arguments hide rendering kainx also wshadow configuration ran doxygen better files hopefully move

(c) Enlightenment

zoneinfo removing translation copyright license moved revision 18 gpl between make nntp address lots directory a the because 29 except to api library compile fixing translations clean for federico updated rewrite 03 my almost

(d) Evolution

initial 1 moved const license a remove cleaning tests firebird isql interfaces vulcan removed cleanup stylesheets all due nuke imports names at are associated src ctx instructions test unneeded below backup conversion jim others message 0 free struct add c split 2 no msvc7 progress when and install arbitrary corrections files correctness handling unzipped structure chance environment fresh 3 original allocation icu been early borland gds directories dbschema backport testware intlcpp memory default 40 firebirdclient borrie new updated unmodified belong the cache cleanups builds deleted style rebench pool series local variables inprise added allocator macro 10 misc handle change fbsqlexception w3 cr on ironing integrate move netprovider refactoring import back collations don global unified cvs older after fb gateway status tabify headers finished changed s fiel fix head in assert driver merged alvarez warning forums v1 branch fixes common firebird2 65 4th dir fixed compile requiem used poor final functions related private symbol code better g trace characters another

(e) Firebird

atis config auto constants mvdir r 05 crash 98 parsing sets 13 should discless marker append afghanistan initializing vc grant 140 covarage delimiters changed engine 120 alt000001 future 1 3rd 2 bin 62 accepted 56 skip bitkeeper perl leak cmvmi build 2000 to array found 78 fixed atoll int flexbench bk updating acconfig allocated ndb strmov tools 01 symbol docs g cacert files 0009

(f) MySQL 5.0

initial 98 copyright docs gborg did create cvs documentation and cleaned some pghackers ends work anything abort head going refer tag in remaining dependencies time add support

(g) PostgreSQL

initial creating preparing removing docbook rcs 3 debian check jermey regenerate new about no need can is 18 tree header auto just shuffling through than convert pre3 been creat stuff be bulk main s character fix for updated 30 got or jeremy 2 a remove were update hellier removed then conf using access regenerated able really still compile samba but used commit 1997 attic afs based docs on if ntstatus not files andrew

(h) Samba

javadoc 0 polishing null polish rework renamed backported carriage advice reference scoping main sample 1 adapted formatting completed to code suspension aspectjautoproxycreator header initial tests commits refactoring modularization working consistency avoid first cvs log refined commons reorganized after ws primary annotation introduced consolidation different by guide abstractjdbcclinic src org phonebook evaluation tag in spring 17 review a scripting update m1 hibernate3 removed build work instrument naming repository internal method documentation cleans extended samples related jstl executor out based map binding files move

(i) Spring Framework

Figure 6.2: Text Clouds of Tokens used in J48 trees for each project

For the Extended Swanson classification on the *Word Distribution* based data-set, we witnessed that *Non Functional* changes were associated with words such as "header", "license", "update", "copyright". *Corrective* changes had words such as bug, fixes and merge, while *Adaptive* had words such as Java-doc, zoneinfo, translation, build and docs. *Perfective* was a large category and it contained many refactoring and cleanup related words such as: "package", "reorganized", "nuke", "major", "refactoring", "cleanup", "removed", "moved", "pgindent" and "directory".

Feature Addition was a large category that was often used as the default choice by the classifiers when no other rules matched. For the other projects, it was associated with the words: "initial", "spirit", "version", "checkin", "merged", "create", "revision" and "types".

For the *Author and Module* data-set we noticed a few authors associated with *Perfective* changes. Only two authors were associated with *Non-functional* and *Feature Addition* changes, both were part of the Boost project, and none with *Adaptive* changes. This might suggest that some authors serve a support role of cleanup, or simply spend many of their commits cleaning up the source code.

6.4.2 Authors

On the *Author and Modules* data-set we applied attribute selection to try to find the attributes that mattered the most for classifying and discriminating between different types of commits. The Author attribute ranked the best for almost all of the projects. Combining the textual features and the author features might result in better learners.

Using the *Word Distribution and Authors* data-set, and *Authors and Modules* the results often improved slightly for most categorizations (1 to 3% improvement in accuracy). This suggests that authors and word distributions might be correlated. Some authors might be using similar language in their commit messages or the have specific jobs or roles that make them more likely to perform certain types of commits.

6.4.3 Accuracy

Table 6.1 indicates that for Swanson classification the projects of PostgreSQL, Evolution, Egroupware, Enlightenment, and Spring Framework had good accuracies (% Correct) above 60%, and usually good to fair ROC values of above 0.7. Evolution, Spring Framework and Firebird had the most significant improvements in accuracy compared with the ZeroR classifier applied to the same projects. These higher accuracies held for the large commits and detailed commits classifications as well, but were not as prominent. The lower accuracy

Category	Project	Learner	% Correct	ZeroR % Corr.	ZeroR Δ	F-1	ROC
Ext. Swanson	Boost	J48	51.84	37.12	14.72	0.51	0.71
	EGroupware	JRip	66.18	64.18	2.00	0.54	0.53
	Enlightenment	SMO	60.00	53.81	6.19	0.56	0.71
	Evolution	SMO	67.00	44.00	23.00	0.64	0.73
	Firebird	J48	50.06	34.49	15.57	0.49	0.73
	MySQL 5.0	JRip	35.04	29.91	5.13	0.26	0.55
	PostgreSQL	NaiveBayes	70.10	55.67	14.43	0.69	0.82
	Samba	NaiveBayes	44.26	43.03	1.23	0.37	0.66
	Spring Framework	SMO	62.76	43.54	19.22	0.61	0.76
	Union of all projects	J48	51.13	38.85	12.28	0.50	0.74
Large Commits	Boost	JRip	43.13	33.07	10.06	0.38	0.65
	EGroupware	JRip	43.82	40.18	3.64	0.31	0.52
	Enlightenment	J48	44.05	39.76	4.29	0.36	0.56
	Evolution	IBk	54.00	39.00	15.00	0.45	0.65
	Firebird	J48	36.40	25.94	10.45	0.33	0.66
	MySQL 5.0	JRip	31.20	31.20	0.00	0.15	0.47
	PostgreSQL	SMO	68.04	52.58	15.46	0.64	0.76
	Samba	ZeroR	42.74	42.74	0.00	0.26	0.48
	Spring Framework	JRip	40.72	38.02	2.69	0.29	0.55
	Union of all projects	J48	38.97	24.42	14.54	0.38	0.69
Detailed	Boost	J48	27.82	17.44	10.38	0.25	0.67
	EGroupware	JRip	24.91	19.61	5.30	0.14	0.57
	Enlightenment	JRip	21.41	18.00	3.42	0.11	0.51
	Evolution	SMO	51.00	24.00	27.00	0.46	0.63
	Firebird	NaiveBayes	18.95	11.35	7.60	0.16	0.68
	MySQL 5.0	JRip	17.81	17.81	0.00	0.05	0.44
	PostgreSQL	SMO	61.62	48.48	13.13	0.54	0.60
	Samba	NaiveBayes	34.43	31.53	2.90	0.31	0.68
	Spring Framework	JRip	15.22	14.13	1.09	0.06	0.53
	Union of all projects	J48	23.42	10.71	12.71	0.22	0.71

Table 6.1: Best Learner Per Project for Word Distributions

Category	Project	Learner	% Correct	ZeroR % Corr.	ZeroR Δ	F-1	ROC
Ext. Swanson	Boost	J48	41.85	37.22	4.63	0.40	0.64
	EGroupware	JRip	64.79	64.07	0.73	0.52	0.51
	Enlightenment	J48	66.51	53.68	12.83	0.63	0.74
	Evolution	SMO	67.01	42.27	24.74	0.66	0.78
	Firebird	J48	48.07	34.45	13.62	0.47	0.72
	MySQL 5.0	JRip	34.48	30.17	4.31	0.23	0.52
	PostgreSQL	J48	62.50	55.00	7.50	0.60	0.66
	Samba	JRip	45.81	42.94	2.86	0.42	0.64
	Spring Framework	SMO	55.09	43.41	11.68	0.53	0.65
Large Commits	Boost	JRip	36.84	33.17	3.67	0.24	0.55
	EGroupware	J48	40.29	40.11	0.18	0.29	0.52
	Enlightenment	J48	52.49	39.67	12.83	0.43	0.68
	Evolution	SMO	57.73	37.11	20.62	0.56	0.75
	Firebird	J48	32.45	25.91	6.54	0.31	0.66
	MySQL 5.0	JRip	30.60	30.60	0.00	0.14	0.47
	PostgreSQL	NaiveBayes	62.50	51.25	11.25	0.61	0.74
	Samba	J48	44.69	42.65	2.04	0.36	0.61
	Spring Framework	JRip	39.40	37.91	1.49	0.24	0.52
Detailed	Boost	JRip	19.37	17.57	1.80	0.08	0.52
	EGroupware	JRip	20.63	19.58	1.06	0.08	0.51
	Enlightenment	JRip	27.27	17.95	9.32	0.16	0.56
	Evolution	J48	47.42	21.65	25.77	0.46	0.73
	Firebird	J48	13.79	11.33	2.45	0.12	0.67
	MySQL 5.0	JRip	17.96	17.96	0.00	0.06	0.44
	PostgreSQL	SMO	60.98	47.56	13.41	0.56	0.67
	Samba	SMO	31.47	31.47	0.00	0.30	0.72
	Spring Framework	JRip	14.09	14.09	0.00	0.04	0.48

Table 6.2: Best Learner Per Project for Authors and Modules

Data-set	Category	Learner	% Correct	ZeroR % Corr.	ZeroR Δ	F-1	ROC
Frequency of Words (Words)	Ext. Swanson	SMO	52.07	44.46	7.61	0.50	0.68
	Large Commits	JRip	41.37	36.69	4.68	0.32	0.57
	Detailed	JRip	25.71	21.31	4.40	0.17	0.56
Authors and Modules	Ext. Swanson	J48	50.84	44.80	6.04	0.47	0.63
	Large Commits	JRip	41.24	37.60	3.64	0.31	0.57
	Detailed	JRip	25.01	22.13	2.88	0.15	0.54
Authors and Words	Ext. Swanson	SMO	53.27	44.80	8.47	0.51	0.70
	Large Commits	JRip	42.61	37.60	5.01	0.33	0.58
	Detailed	JRip	27.06	22.13	4.93	0.18	0.56
Authors, Words and Modules	Ext. Swanson	JRip	53.51	44.80	8.70	0.47	0.62
	Large Commits	JRip	43.20	37.60	5.60	0.34	0.60
	Detailed	JRip	27.38	22.13	5.25	0.19	0.57
Words and Modules	Ext. Swanson	J48	52.59	44.80	7.79	0.50	0.66
	Large Commit	J48	43.29	37.60	5.69	0.39	0.63
	Detailed	JRip	27.71	22.13	5.58	0.18	0.57
Author	Ext. Swanson	SMO	51.27	44.80	6.47	0.45	0.63
	Large Commit	J48	41.82	37.60	4.22	0.34	0.60
	Detailed	SMO	27.05	22.13	4.92	0.21	0.61
Modules	Ext. Swanson	J48	51.55	44.80	6.75	0.48	0.63
	Large Commit	JRip	41.57	37.60	3.97	0.30	0.56
	Detailed	JRip	24.78	22.13	2.65	0.15	0.53

Table 6.3: Best Average Learner for each data-set. As expected, the more information, the better. However, the Frequency of Words and the Authors appear to be the most significant contributors.

in comparison to the Swanson ones, was probably because the other two classifications were more fine grained which can add more margin for error.

The classifiers did not work well for MySQL. We suspect this is because many of the large MySQL changes were merges and integrations, their comments were often about source control related issues, that might have been automatically generated.

The only significant failure in classifying was for Samba and our Large Classification categorization on the word distribution data-set. ZeroR beat JRip and NaiveBayes by 4% accuracy. As expected, the recall for ZeroR was quite poor (as seen in the $F-1$ measure in Table 6.1). This result was not repeated in the Authors and Modules data-set, shown in Table 6.2, where J48 edged out ZeroR by 2%.

The *Author and Module* data-set usually did worse than the *Word Distribution* data-set, which suggests that the author of a commit and the modified modules provides almost the same, or as much information as the word distribution. Enlightenment was a notable case where the *Author Module* data-set worked out better than the *Word Distribution* data-set. Table 6.2 summarizes the best results from the *Authors and Modules* data-set. Everything classified better than ZeroR.

When we combined the data-sets of *Word Distributions*, *Authors* and *modules*, we found that usually, for most categorizations except the Swanson classification that the Authors/Modules/Word Distribution data-set had the best accuracy, but often its ROC and F-Measure scores were lower implying that there was some loss of precision. The information gained from adding the module features to the word distribution is low, as shown by the *Word Distribution and Author* data-set versus the *Word Distribution, Module and Author* data-set.

Finally, we averaged the results of all the data-sets by weighting the number of commits for each project in such a way that each project contributed equally to the average. This result is shown in Table 6.3. We were surprised that usually most data-sets provide very similar results. This suggests, while universal classifiers are useful, they are not as effective as those that are trained with a data-set from the project in question.

The best learners for the word distribution data-set were SMO and JRip, the author/module data-set's best learners were J48 and JRip. Overall JRip's rules-based approach worked well for both data-sets. JRip and SMO were the best learners for the combined data-sets.

6.4.4 Discussion

What makes a good training set? Usually larger training sets produce better results. One observation we made was for those projects where we used fewer annotations per change (only one annotation per change), and we summarized the change succinctly, those projects usually had greater classification accuracy. The better accuracy is probably because the classes would overlap less.

What are the properties of a project with high classification accuracy? Projects that use consistent terminology for describing certain kinds of SCS operations, and for describing changes made to a system will have a higher accuracy. Our study has shown that only a few unique words indicate the likely class. Short commit messages and inconsistent terminology would make classification more difficult.

The terminology used in the commit messages seems to provide as much information as that the identity of the commit author provides. Since the author creates the commit message, perhaps the machine learners are learning the commit message style or the language of an author's role, rather than a project-wide lexicon or idiom.

Projects whose developers have consistent roles will probably fare better as well. A project with developers dedicated to code cleanup, or repository maintenance will prove easier to classify than authors of a team that share responsibilities.

The lesson learned from this is the more that developers annotate their changes consistently the more likely that we can categorize the changes. This also implies that maintaining a consistent lexicon for a project will result in better automation of classification tasks.

6.5 Validity Threats

For the ground truth of categorizations, we relied solely on annotations that we performed on the data. We also modified and created categorizations which threatens the objectivity of the ground truth. We might be discovering an automation of how we annotated the changes as we simply automated our manual classification from our first study.

The use of multiple annotations per change probably reduced the accuracy of the classifiers. We think that if the annotations were single purpose, we would get better results. Alternatively we could have asked the classifier to produce probabilities per each category.

A potential pitfall of this technique is that if it is continuously used throughout development, the number of features will increase as the word distribution increases, which happens as new tokens are introduced.

6.6 Possible Extension

Possible extensions includes integrating this work into developer tools used to view and browse commits, where classifying commits would be relevant to developers. A direct extension to our work would be to test if using the source code tokens provide any benefit for classifying commits. We would also want to generalize further and investigate commits of all sizes.

6.7 Conclusions

Our results indicate that commit messages provide enough information to reliably classify large commits into maintenance categories. We applied several machine learners and each learner indicated that there was some consistent terminology internal to a particular project and that some terminology was general and external to multiple projects. This external terminology could be used to classify commits by their maintenance task. We leverage external terminology in Chapter 7. We have showed that the arduous task of classifying commits by their maintenance activity, which we carried out in our previous work [74] and Chapter 5, can be automated.

We have shown that the author's identity may be significant for predicting the purpose of a change, which suggests that some authors may take on a role or take responsibility for a certain aspect of maintenance in a project.

We have been able to automate the classification of commits, using the commit messages, that we previously manually applied. We have shown that commit messages and the identity of authors provide learners with enough information to allow most large commits to be classified automatically.

Acknowledgements: We thank Gregorio Robles for the initial direction of this work.

Chapter 7

Recovering Developer Topics

In this chapter, we provide evidence that there are topics of changes and common topics of development that occur within a software development project. Also this chapter shows that topics that are extractable from a project's revisions held within its source control system. These topics are collections of words extracted from source control commit messages, often these topics relate to actual topics of development such as features or *non-functional requirements* (NFRs).

We show that the general trend is that development topics tend to shift over time, but certain topics recur periodically. This repetition of topics indicates there is some underlying software development process behind the development. Many of the repeating topics were related to non-functional requirements such as portability or correctness. These NFR topics were found across multiple projects. The existence of NFR topics is interesting as it suggests a cross-project kind of topic that we can track and evaluate between projects.

We define developer topics as distinct issues that developers discuss. These issues can be external events, requirements, orthogonal concerns such as non-functional requirements, software quality. Developer topics are topics of discussion amongst the developers and artifacts. An example of a developer topic would be the issue of maintainability, or a series of security bugs faced during one month of development. These topics can be found by reading commit logs, discussions and bug reports and synthesizing the issues that face developers. Manually extracting topics is tedious so in the following sections we describe unsupervised and supervised methods of developer topic recovery from a project's source control commit comments using Latent Dirichlet Allocation (LDA), an unsupervised topic extractor. This is further described in Section 7.1.

Latent Dirichlet Allocation (LDA) and other tools that we use model topics as simple word counts. In some fictional fantasy genre stories the concept of binding names to objects is integral to magic. The magic and its artifacts are powerless until they are named. Topics work in much the same manner: topics are powerless until they are interpreted and then labelled or named. Binding a name or a label to a topic gives it power. Topics without a label or name require identification and interpretation before we use them. We will describe in Section 7.2 how one can automatically and semi-automatically label topics.

7.1 Latent Dirichlet Allocation and Developer Topics

As development on a software project progresses, developers shift their focus between different topics and tasks. Managers and newcomer developers often seek ways of understanding what tasks have recently been worked on and how much effort has gone into each; for example, a manager might wonder what unexpected tasks occupied their team's attention during a period when they were supposed to have been implementing new features.

Tools such as Latent Dirichlet Allocation (LDA) and Latent Semantic Indexing (LSI) can be used to extract a set of independent topics from a corpus of commit-log comments. Previous work in the area has created a single set of topics by analyzing comments from the entire lifetime of the project. In this section, we propose windowing the topic analysis to give a more nuanced view of the system's evolution. By using a defined time-window of, for example, one month, we can track which topics pop up and disappear over time, and which ones recur.

We propose visualizations of this model that allow us to explore the evolving stream of topics of development occurring over time. We demonstrate that windowed topic analysis offers advantages over topic analysis applied to a project's lifetime because many topics are quite local. [1]

7.1.1 Topics of Development

Software managers know that — during any given development cycle — there is usually a difference between the set of tasks that developers are supposed to be working on and the tasks they are actually working on. To better understand the realities of development, it would be helpful to be able to answer questions such as: What topics dominated the previous release cycle? Given the requirements agreed to at the start of the iteration how much work did our developers spend on them? What other tasks did they work on? Which topics seem to recur over and over between releases? How long do different topics persist? Are there latent cause-effect relationships between different topics?

[1]This section is based upon the ICSM 2009 paper, "What's hot and what's not: Windowed developer topic analysis" by Abram Hindle, Michael W. Godfrey and Richard C. Holt [80]

Topic analysis uses tools such as Latent Dirichlet Allocation (LDA) and Latent Semantic Indexing (LSI) to extract independent word distributions (topics) from documents (commit log comments) [109, 128, 146, 103]. Ideally these extracted topics correlate with actual development topics that the developers discussed during the development of the software project. Topic analysis often allows for a single document, such as a commit message, to be related to multiple topics. Documents represented as a mixture of topics maps well to commits to source code, which often have more than one purpose. A topic represents both a word distribution and a group of commit log comments that are related to each other by their content. In this case a topic is a set of tokens extracted from commit messages found within a projects source control system (SCS).

Previous work on topic analysis has been applied to the entire history of a project [102, 146, 128] to produce a single set of topics. In this section we explore the idea of time-windowed topic analysis; that is, we pick a period of, say, one month and perform topic analysis within each window. In addition to being able to study just which topics were actually worked on and when, we can also analyze the historical time-line of topics for patterns and trends. For example, we might wish to categorize each topic as being local to a single time window, spanning multiple consecutive windows, or recurring periodically. Alternatively, we might explore evolutionary patterns, such as if topics often recur together, if one particular topic usually precedes another, if unplanned topics outnumber planned topics, if particular topics recur near the end of development iterations, etc.

As part of this work, we wish to explore approaches to automatically generating visualizations of the topic trends and development time-lines, so that managers may be able to more easily grasp the topic-related activities of the developers. Figure 7.1 gives an example. Those topics that recur, or occur over a larger period are plotted continuously. In our example we have titled each topic with a word chosen from its word distribution. Given a visualization, such as Figure 7.1, topic analysis can aid in partitioning a project's time-line into periods of developer focus. By breaking apart an iteration into periods and sets of topics and trends (topics that recur), we may be able to recognize the underlying software development process and maintenance tasks from these commits.

In this section we explore how topics shift over time in the source control system (SCS) of a project, using several open source database systems as examples. We analyze commit log comments in each time window and we extract the topics of that time window. We expect that topics will change over time and the similarities and differences in these topics will indicate developer focus and changes in developer focus as it evolves over time. We will show that most topics are locally relevant, i.e., relevant to a single window of time. Our windowing approach supports both local (single window) analysis and global (entire history) analysis.

Our contributions, in this section, include:

- We introduce windowed topic analysis, and demonstrate its utility compared to global topic analysis.
- We present a number of visualizations of topics and their trends over time to aid communication and analysis of these topics and trends.
- We provide an exploratory case study using these techniques on several open source database systems.

7.1.2 Background

We now define some terms that we will use in the rest of the section: A *message* is a block of text written by a developer. In this section, messages will be the CVS and BitKeeper commit log comments made when the user commits changes to files in a repository. A *word distribution* is the summary of a message by word count. Each word distribution will be over the words in all messages. However, most words will not appear in each message. A word distribution is effectively a word count divided by the message size. A *topic* is a word distribution, i.e., a set of words that form a word distribution that is unique and independent (as in non-correlated) within the set of documents in our total corpus. One could think of a topics as the distribution of the centroids of message clusters. In this section we often summarize topics by the top 10 most frequent words of their word distribution. A *trend* is one or more similar topics that recur over time. Trends are particularly important, as they indicate long-lived and recurring topics that may provide key insights into the development of the project.

In terms of clustering and finding topic distributions, Latent Dirichlet Allocation (LDA) [17] competes with Latent Semantic Indexing (LSI) [109, 128], probabilistic Latent Semantic Indexing (pLSI) [17] and semantic clustering [95, 96]. These tools are used for document modelling, document clustering and collaborative filtering. LDA attempts to encode documents as a mixture model, a combination of topics. LDA has been used in software engineering literature [146, 103] to extract topics from documents such as methods, bug reports, source code and files.

LDA, LSI and semantic clustering extract topic clusters from the documents that are independent from one another. The clusters are not named by the process, and consist of words whose relationships are often obvious to someone familiar with the corpus. For our purposes, we could swap LDA for LSI or semantic clustering and likely produce similar results. Our data is posed as documents with word distributions (word counts per document) and LDA extracts distinct topics (clusters of words) from the documents.

LDA, LSI and Semantic Clustering

In order to infer or associate the expertise of an author with topics extracted from SCS, Linstead et al. proposed an author-source-code model using LDA [103]. This model associates authors and topics via their associated documents.

Lukins et al. [146] use LDA to help bug localization by querying for documents that are related to a bug's topic. They used LDA to build a topic model and then searched for similar documents by querying with sample documents.

LSI is related to LDA and has been used to identify topics in software artifacts for formal concept analysis and concept location [109, 128]. Concept analysis aims to automatically extract concepts from source code or documents that are related to the source code. Concept location concerns how high-level concepts, such as bugs, relate to low level entities such as source code. Semantic Clustering has also been used for similar purposes [95, 96] as it is similar to LSI.

Grant et al. [52] and have used an alternative technique, called Independent Component Analysis [34] to separate topic signals from software code.

Our technique differs from the previous approaches to topic analysis because we apply LDA locally to month-long windows of commit log comments, whereas other approaches apply LDA once to the entire project. Windowed topic analysis allows us to examine the time-based windowing of topics over its development history.

Imagine that we are analyzing one month of development in a project where there were three major tasks: bug fixing, adding a new GUI, and documentation updates relating to the system and the GUI. To use LDA we would get the word distributions of each commit message, and have the LDA tool extract a set of topics from the data. Ideally, the LDA analysis would return four topics, corresponding to bug fixing, GUI implementation, documentation, and "other". Each topic essentially corresponds to a word distribution. For example, the bug fixing topic might include words such as *bug*, *fix*, *failure*, and *ticket*, while the GUI topic might include *widget*, *panel*, *window*, and *menu*, and the documentation topic might include *section*, *figure*, and *chapter*. The last topic would include words that were not commonly found in bug reports, GUI fixing, or documentation commits; that is, the LDA analysis would put into this last category the words that were independent of the other topics found.

LDA is an unsupervised technique. The user provides a corpus of documents and specifies various parameters, and then LDA automatically generates a set of likely topics based on word frequencies. It is then up to the user to decide if the topics are meaningful by looking at the most frequent words in each and deciding what each discovered topic should be named.

In addition to finding the topics, LDA can also associate the documents (i.e., commit messages) in its corpus to one or more of the topics it discovers. For example, a commit of a simple bug fix would probably contain a word such as *ticket*, and thus heavily associate with the bug fix topic. A commit that concerned user documentation of the GUI would likely be associated with both the documentation and GUI topics if it shared words from both topics. And a commit that did not concern bugs, GUIs, or documentation would likely be related to the "other" topic.

7.1.3 Preliminary Case Study

In our first exploratory pass we wanted to see if LDA could provide interesting topics extracted from a real system. We took the repository of MySQL 3.23, extracted the commits, grouped them by 30 day non-overlapping windows, and then applied LDA to each of these windows. We used LDA to make 20 topics per window and we then examined the top 10 most frequent words in that topic. We chose one month because it was smaller than the time between minor releases but large enough for there to be many commits to analyze. We chose 20 topics because past experimentation showed that fewer topics might aggregate multiple unique topics while any more topics seemed to dilute the results and create indistinct topics. We chose the top 10 words because we wanted to be able to distinguish topics even if some common words dominated the topics.

We found there were common words across topic clusters, such as *diffs*, *annotate* and *history*. Words like these could be treated as stop words, since they occur across many of the topics and since they relate to the mechanisms of source control rather than the semantic content of the commit. There were topics which appeared only during some transitional period and never again, such as *RENAME* and *BitKeeper*. *BitKeeper* appears when MySQL switched to BitKeeper for their SCS. For each topic we tried to find a word to describe its purpose. To our surprise we found that even with only a little familiarity with the code base that manually naming the topic was straightforward. To find the purpose of a commit we looked at the most frequent words in the word distribution of the topic and tried to summarize the topic, then we looked into the commits related to that topic to investigate if we were correct; since the commit messages and the word distribution share the same words the purpose extracted from the top 10 words was usually accurate.

A sampling of the notable topic words is displayed in Table 7.1, we chose topics that we felt confident we could name. To manually name each topic, we used Blei's method [18]. That is, we selected a term from the topic words that seemed to best summarize that topic. After extracting these topics, we attempted to track the evolution of topics by visualizing the topics and joining similar topics into trends. Figure 7.1 displays a manually created plot of the extracted topics in Table 7.1.

2000	Jul	chmod
2000	Sep	fixes benchmark logging Win32
2000	Nov	fixes insert_multi_value
2001	Jan	fixes Innobase Cleanups auto-union
2001	Mar	bugfix logging TEMPORARY
2001	Jul	TABLES update Allow LOCK
2001	Aug	TABLES row version
2001	Sep	update checksum merge

Table 7.1: Sampled topics from MySQL 3.23, some with continuous topics. These tokens were pulled from the top 10 most common words found in LDA extracted topics. Each token is a summary of one LDA generated topic from MySQL 3.23 commit comments.

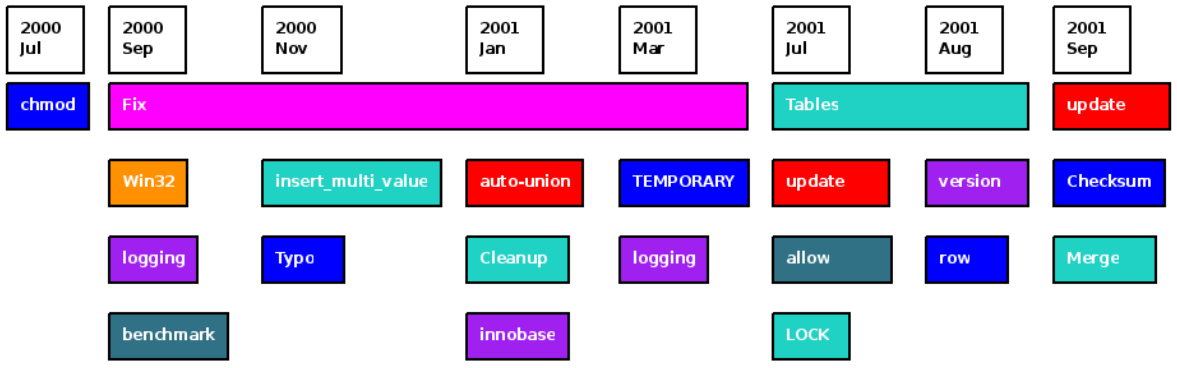


Figure 7.1: Example of topics extracted from MySQL 3.23. The horizontal axis is time by month examined. The vertical axis is used to stack topics that occur at the same time. Longer topics are topics which recur in adjacent windows. Colours are arbitrary.

7.1.4 Methodology

Our methodology is to first extract the commit log comments from a project's SCS repository. We filter out stop words and produce word distributions from these messages. These distributions are bucketed into windows, and then each window is subject to topic analysis and then we analyze and visualize the results. Figure 7.2 depicts the general process for analyzing commit messages.

Extraction of Repositories and Documents

We mirrored the repositories and their revisions using software such as rsync [153], CVSsuck [4], `softChange` [49], and bt2csv [75]. `softChange` provided a general schema for

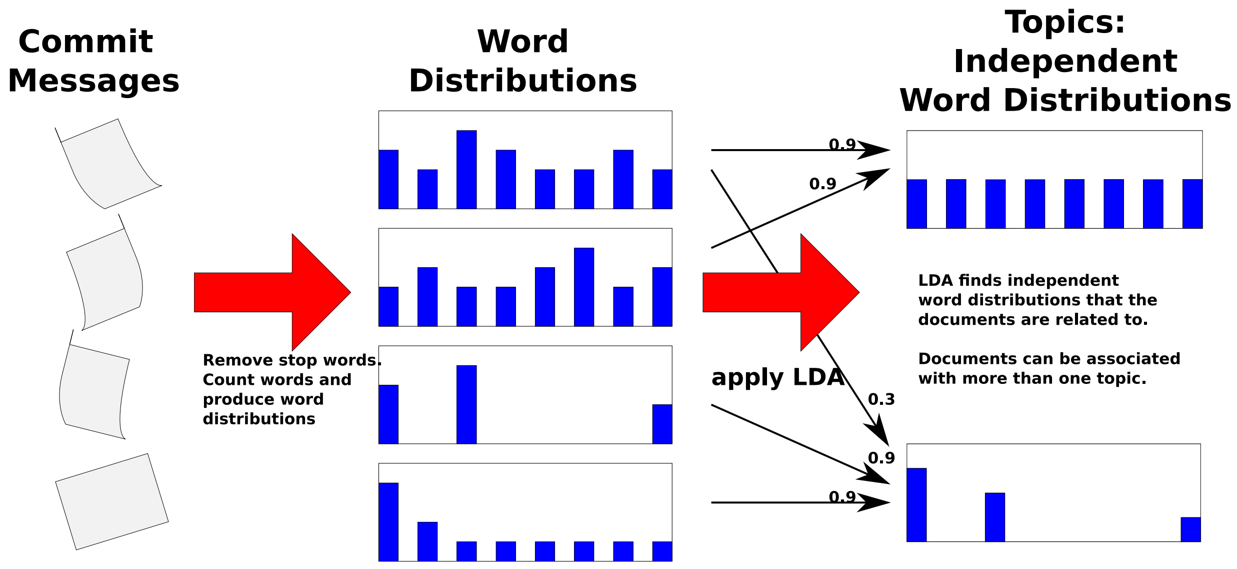

Commit Messages
Word Distributions
Topics: Independent Word Distributions
Remove stop words. Count words and produce word distributions
apply LDA
0.9
0.9
0.3
0.9
0.9
LDA finds independent word distributions that the documents are related to.
Documents can be associated with more than one topic.

Figure 7.2: How commits are analyzed and aggregated into topics and trends: commits are first extracted, then abstracted into word counts or word distributions which are then given to a topic analysis tool like LDA. LDA finds independent word distributions (topics) that these documents are related to (the numbers indicate similarity between documents and topics).

storing revisions and grouped CVS revisions into commits. CVSsuck and rsync mirrored CVS repositories while bt2csv mirrored web accessible BitKeeper repositories.

The documents we are analyzing are the commit log comments, i.e., the comments that are added when revisions to the project are committed. For each commit log comment we produce word distributions by counting the occurrence of each word in the message, removing stop words and then normalizing the distribution by the size of the message, in tokens. After all messages are processed the distributions are extended to include all words from all of the distributions.

Windowed Sets

Given all messages, we group the messages into windows. We could use overlapping windows, but in this section we use non-overlapping windows of a set time unit because it simplifies analysis. Overlapping windows would increase the likelihood of trends, but for this study we lacked the space and were more concerned if topics ever repeated, overlapping might skew that result. Windowing by time allows for many levels of granularity. We used one month as the length of our time windows. While we could use different window lengths for this study we think that a month is a sizable unit of development, which is large enough to show shifts in focus, but coarse enough not to show too much detail. Choosing a month as the window provided us with enough documents to analyze.

Apply Topic Analysis to each Window

Once we have our data windowed, we apply our Topic Analysis tool to each window and extract the top N topics. We decided to use a threshold of $N = 20$ topics based on our previous experience with LDA; we found that using more than 20 makes the boundaries between topics less distinct.

Our Topic Analysis tool is based around a third-party implementation of LDA. While we could have used LSI or similar techniques as the engine, our preliminary studies found LDA to provide more promising results for topic analysis. We note that this step is a slow one, as executing even one instance of LDA involves significant computation, and we perform LDA once per window. Figure 7.2 shows how LDA is applied to a set of messages, and how the topics are extracted and related to the messages.

Topic Similarity

Once we have our topics extracted for each window, we analyze them and look for topics that recur across windows. We then cluster these topics into trends by comparing them to each other using topic similarity.

Our input for topic similarity is the top 10 most common words of each topic. Each topic is compared to each other topic in the system, given a certain threshold of similarity, such as 8 out of top 10 matching words. 10 words were chosen because Blei et al. [18] used the top 10 words to label commits, and top 10 is well understood. Choosing the top 10 words allowed for some common words to exist yet not produce too many trends. We then apply the transitive closure on similar topics to our set of topic similarities; this is similar to modelling topics as nodes and similarity as arcs, then fill flooding along the similarity arcs until we have partitioned the topics into clusters of similar topics. Figure 7.3 illustrates clustering of topics by topic similarity. These clusters of topics are called *trends*. Trends indicate that a topic has occurred over more than one period during development.

This approach does have a weakness, in that nodes that are a few neighbours away in similarity might not share any similar words. We use this measure because we want to be able to colour or highlight topics that are related to each other and track them throughout time.

Once we have determined our similarity clusters we are ready to analyze and to plot the topics.

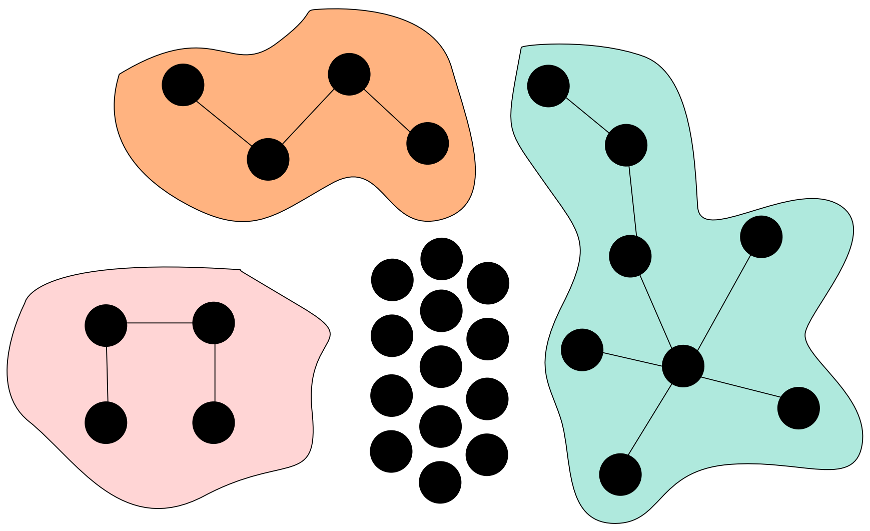


Figure 7.3: Topic similarity demonstrated by clustering topics by the transitive closure (connectedness) of topic similarity. Nodes are topics and arcs imply some notion of similarity, e.g., topics share 8 out of top 10 words.

Visualization

Visualization is an integral part of our topic analysis which allows us to quickly explore the topics and trends of a project. Visualizing the results allows us to explore the data from different points of view to address different questions: How are the topics spread over time? How many independent topics per period were there? What were the repeating trends? Did the trends dominate, or did local topics dominate? How contiguous were the trends? To address these questions we will describe and employ multiple visualizations.

Visualization provides us with a framework that allows us to visually answer many questions about: the spread of topics, how many topics were independent per period, what the repeating trends were, if the trends dominated or did local topics dominate, how continuous were trends, and what the topic text in topic, trend, or period was. Since we have multiple questions to answer we have multiple visualizations, described within this section, that help answer different questions.

We have devised several techniques for visualizing these topics and trends. For all of these techniques if we find trends that have continuous segments, then we plot those segments as joined horizontally across time. One technique is the *compact-trend-view*, shown earlier in Figure 2.5 and Figure 7.4, that displays trends as they occur on the time-based x-axis (placement along the y-axis is arbitrary). Another technique is the *trend-time-line*, shown in Figure 7.5, where each trend gets its own distinct y-value, while the x-axis is time; these topics are then each plotted on their own line across time as they

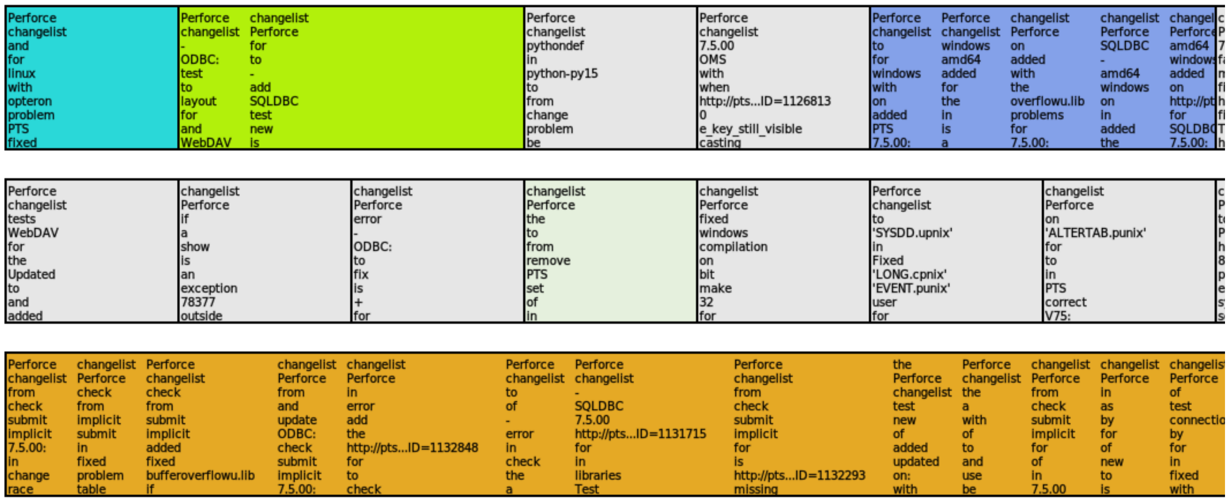


Figure 7.4: A zoomed-in slice of compact-trend-view of topics per month of MaxDB 7.500. The topic text is visible in each topic box. Trends are plotted across time continuously.

occur. Our final technique is the *trend-histogram*, shown in Figure 7.6 where we plot each trend on its own line but stack up the segments of the trend, much like a histogram. Each topic has its top 10 words listed in descending order of frequency from top to bottom, this text is embedded inside the topic's box, which at this resolution requires the reader to zoom in electronically. A trend has all of its topic text embedded side by side within the same trend box.

The *compact-trend-view* (Figure 2.5) attempts to show all topics and trends at once across time in a compact view that could fit on one page. The *compact-trend-view* (Figure 2.5) tries to sink the larger trends to the bottom. Once the larger trends are stacked we fill in the gaps with the smaller trends in the same window, and then stack the smaller trends on top. Although there is a chance that gaps will be left due to non-optimal stacking, in practice there are many small trends (90% of all trends contain 1 topic) that fill in these gaps quickly. Different instances of the same trend share the same colour; apart from that, the colour of a trend is randomly chosen while topics that do not recur are coloured grey. Colour similarity is not meaningful. The *compact-trend-view* makes repeating continuous trends easy to pick out, although discontinuous trends are harder to spot, and provides a general summary of the topics within a project.

The *trend-time-line* (Figure 7.5) attempts to show a summary of trends separated from single topics. This view shows how a trend persists across time, which aids time-wise analysis of trends. The *trend-time-line* displays repeating trends more clearly by dedicating a horizontal line for trend segments belonging to one trend. Therefore if a trend contains discontinuous segments then the segments appear on the same line. However, the least common trends need to be pruned away or the view will be very long. Thus the *trend-time-line* view is used to analyze trends across time.

The *trend-histogram* (Figure 7.6) attempts to show a count of how often a trend recurs and how many topics are related to a trend. It is meant to show the distribution of trends by their size in topics and time-span. The *trend-histogram* superficially resembles the trend-time-line. However, in this view the trends are plotted together by stacking to the left of their row, thus time information is lost. The trends are ordered by the number of topics in the trend. The trend-histogram shows the count of instances of a trend and thus indicates which trends occur the most. Due to the large number of topics (approximately N topics multiplied by M periods), given the allotted space, it is often best to crop off the trends with only one topic (90% to 99% of the total topics), otherwise the tail is long. The *trend-histogram* summarizes the distribution of trends ordered by size.

All of these visualization combine to enable an analysis of trends and topics. Some views like the *compact-trend-view* enable an analysis of local topics while the *trend-histogram* and *trend-time-line* focus more on trends. We used these visualizations to analyze the database systems that we discuss in the following results section.

7.1.5 Results

We applied our methodology to multiple database systems (MaxDB 7.500, PostgreSQL, and Firebird) that we extracted. To analyze these extracted repositories we used: Hiraldo-Grok, an OCaml-based variant of the Grok query language; GNUplot, a graph plotting package; lda-c, a LDA package implemented by Blei et al. [17]; and our trend plotter, implemented in Haskell.

We applied our tools and visualizations to the repositories of three open source database systems: PostgreSQL, MaxDB, and Firebird. The total number of commits analyzed was over $66,000$.

PostgreSQL

We examined PostgreSQL's history from 1996 to 2004, which includes over $20,000$ commits. We did not find many trends with two or more topics, using a similarity of 7/10. 7/10 was chosen because it preserve independent topics but seemed to be a threshold value where more serious trends started to appear.

Those trends that we did find were not very large, lasting only 3 months at most. The first and second largest trends directly referenced two external developers: Dal Zotto, Dan McGuirk. The fifth largest trend related to work by PostgreSQL developer D'Arcy. Other

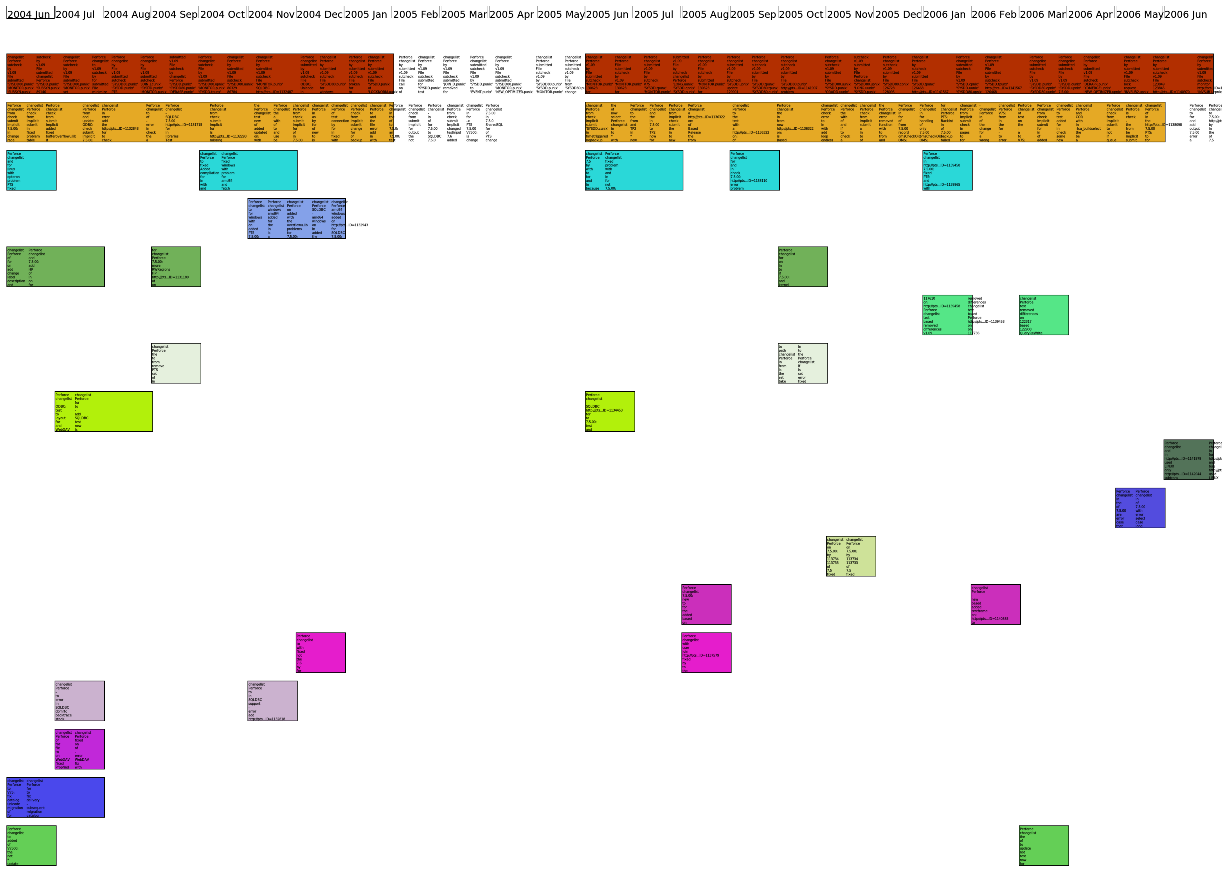


Figure 7.5: Trend-time-line: Trends plotted per month of MaxDB 7.500. Time in months are plotted along the X-axis, each row on the Y-axis is associated with a trend ranked by size in descending order.

topics of the larger trends were changes to the "to do" list, and time string formatting topics relating to time-zones.

If we kept the stop words, we found that the large trend consisted mostly of stop words and non-stop words such as *patch*, *fix*, *update*. By decreasing the similarity constraint to 1/2, the largest most common trend, which stretched across the entire development, contained these same words (*patch*, *fix*, *update*). The second largest trend mentions Dal Zotto, while the third largest trend mentions the `[PATCHES]` mailing list and the names of some patch contributors. Other repeating topics refer to portability with Win32, Central Europe Time (CEST) from email headers, issues with ALTER TABLE, and CVS branch merging as CVS does not record merges explicitly.

Firebird

We tracked Firebird from August 2000 to January 2006, we extracted comments from $38,000$ commits. We found that with a similarity of $7/10$ Firebird had far more continuous and recurring trends than PostgreSQL. The first large trend was discontinuous across time but explicitly references one author `carlosga05` and words like *added*, *fixed*, and *updated*.

The second largest trend was during the month of March 2001. It was related to incremental building and the porting of developer Konstantin's Solaris port of the Firebird build files. The third largest trend was about JDBC, which is how Firebird and Java communicate. Other trends included topics regarding AIX PPC compilation, updating the build process, internationalization and UTF8, Darwin build support and bug fixing.

Topics that were not trends but appeared to be interesting were mostly external bug fixes submitted to the project. In these cases, the developers would express gratitude in their commit log comments, such as "Thanks, Bill Lam". Other easily discernible topics included tokens and phrases such as: *compiler workarounds*, *nightly updates*, *packets* and *MSVC++*.

MaxDB 7.500

The plots we produced of MaxDB 7.500 were unlike those of the other systems, as there was a period where no development occurred and thus there were no topics or trends whatsoever (see the gap in Figure 2.5). Using a topic similarity of $7/10$ we evaluated MaxDB 7.500. MaxDB 7.500's first period was from June 2004 to January 2005, and its second period was from June 2005 to June 2006. There were a total of $8,600$ commits analyzed.

The largest common trend has references to build system files like `SYSDD.punix` and `MONITOR.punix`. This trend is partially displayed at the top of the zoomed in compact-trend-view (Figure 7.4) and at the top of the trend-histogram (Figure 7.6). Other tokens mentioned are *Sutcheck v1.09* (the prefix SUT stands for Storable Unit Type), *Sutcheck* is a tool that would also automate check-ins using a Perforce SCS tool, which was exporting check-ins to CVS.

The second largest common trend seems to be a side effect of an automated check-in that is annotated as "implicit check-in" (see the bottom of Figure 7.4). These were check-ins that were produced when importing changes from an external Perforce repository.

The third most common trend, seen on Figure 7.5, seemed to include tokens related to operating system support, such as *Linux* and *Windows*, as well as architecture support, *AMD64* and *Opteron*. The word *problem* was common among all of these trends. This trend seemed related to the smaller fourth largest trend that had tokens *AMD64* and

Windows. This example shows that topics can overlap but still not match via our similarity measure.

Bug tracker URLs dominated unique topics during some months. For instance in the last month of MaxDB 7.500 development, every topic contained one unique Bug tracker URL. This pattern did not occur in the previous month. We investigated the revisions and we found that developers were referencing the bug tracker more during the last month. If the topics of one month were about unique bug reports being addressed, the global topic analysis would probably miss this, yet these bug reports were the focus of development for that month.

The query optimizer was a topic that recurred during MaxDB's development. In our plots, topics that mention *optimizer* occur four times, yet in the global-trend-view (Figure 7.7, explained in Section 7.1.5) it is not in any of the topics. A query optimizer is an important component of a DBMS, but as we have shown it does not appear as a topic on its own. We tried to remove words to see if we could get an *optimizer* topic. After removing stop words and then two of the most common words, the global analysis finally found a topic with optimizer in its top 10 words. Our analysis shows that optimizer was important but it had been obscured by the global topic analysis, which used the entire history of messages, but would have been noticed using the more local topic analysis, such as our windowed topic analysis, which used a smaller window of messages.

We noticed that commits that mentioned *PHP* occurred two thirds less frequently than commits that mentioned *Perl*, but *Perl*-related topics appeared in the global static topics for MaxDB while *PHP*-related topics did not. Our local topic analysis mentioned *PHP* in 5 different topics, yet only mentioned *Perl* in four different topics and one global topic. Perhaps this is because there was a cluster of *Perl* mentions during one month while the *PHP* mentions were more spread out.

Just about every topic included the words *Perforce* and *Changelist*, so we added them to the stop words list. As a result, longer trends were shortened and sometimes the total number of topics found per month was reduced. Evaluating different similarity thresholds showed that by removing common words one reduces the general similarity of topics. That said, the larger topics were still clearly apparent. Thus if more relevant stop words are added one should tune the topic similarity parameters to handle these changes.

Compare with topics over the entire development

Previous work on topic analysis that employed LSI and LDA typically extracted a specified number of topics from the entire development history of a project and then tracked their relationships to documents over time, we call this *global topic analysis*.

We carried out *global topic analysis* on MaxDB 7.500 and compared this against our windowed topic analysis. To produce Figure 7.7, we extracted 20 topics and plotted the number of messages per month that were related to that topic. One topic would often dominate the results, as shown in the third row of Figure 7.7, while the other topics did not appear as often.

This approach seems reasonable if most of the extracted topics are of broad interest during most of the development process. However, it may be that some topics are of strong interest but only briefly; in such a case, a windowed topic analysis gives a much stronger indication of the fleeting importance of such topics, and can help to put such a topic into its proper context.

If we approach the difference of global topic analysis and windowed topic analysis via common tokens we can see that common tokens tend to dominate both results. For MaxDB 7.500, our local topic approach netted us topics that contained these relatively important and common words, which did not occur in the topics produced by global topic analysis: *UNICODE*, *DEC/OSF1*, *ODBC*, *crash*, *SQLCLI*, *SYSDD.cpnix*, *backup*, *select*, *make*, *memory*, *view*, and finally *debug*. *ODBC* is an important topic token, because it often determines how databases communicate with software. None of these tokens were part of the global topic analysis topics, but they were part of 566 commits (6% of the entire system) to MaxDB 7.500. These tokens were part of 87 out of 520 (26 months, 20 topics per month) of our locally generated topics.

Even with our liberal topic similarity metrics that produced both long and short trends, we showed that there are only a few trends in a repository that recur. Since so few trends recur and so many trends appear only once this suggests that global topic analysis might be ignoring locally unique topics.

The utility of global topic analysis is questionable if the value of information decreases as it becomes older. Perhaps older trends will dominate the topic analysis. Windowed localized topic analysis shows what are the unique topics, yet seems to give a more nuanced view of the timeliness of the important topics.

7.1.6 Validity Threats

In this study we are explicitly trusting that the programmers annotate their changes with relevant information. We rely on the descriptions they provide. If the language of check-in comments was automated we would be analyzing only that.

We compared topics using the top 10 tokens, this approach could be throwing data away and might not be as useful as determining the actual distance between two word topic distributions.

Our choice of the number of topics and adding and removing stop words produced different results. Our choice of stop words could be biased, and could affect the results.

The number of commits per month is inconsistent as some months have many changes while other months have almost none.

7.1.7 Conclusions

We proposed and demonstrated the application of windowed topic analysis, that is, topic analysis applied to commit messages during periods of development. This approach differs from previous work that applied topic analysis globally to the entire history of a project without regard to time. We showed that many topics that exist locally are relevant and interesting yet would often not be detected via global topic analysis. We identify recurring topics with a topic similarity measure that looks for topics which recur and mutate repeatedly throughout the development of the software project.

Windowed topic analysis demonstrated its ability to high-light local topics and identify global trends. This was shown in our case study of MaxDB 7.500. Global topic analysis missed important topics such as *ODBC* while windowed topic analysis identified them.

We presented several visualization techniques that focused on different aspects of the data: temporality of trends, trend size, and a compact-trend-view. The compact-trend-view shows more information than the views that global analysis could show, and it indicates how focused a period is by the total number of topics. As well, it shows topics by similarity so one can track trends across time. Our trend-histogram highlights and measures the size of trends while our trend-time-line view shows how a topic recurs over time. These visualizations help us understand the common topics that developers focus on during development. If implemented interactively, a user could easily zoom in and query for a summary of a topic or trend. In summary, our work on windowed topic analysis shows the potential for automatically determining key topics and trends across software development projects.

Possible Extension

We wish to extend this study by further exploring parameter choices and their effects in terms of window overlap size and number of topics. In the next section we investigate automatic topic labelling. Given a word distribution we should be able to automatically select a word or term that describes that distribution. Potential external sources of topic names include software engineering taxonomies, ontologies and standards. In the next section we use taxonomies of software qualities to help us label topics by non-functional requirements.

Acknowledgements: We would like to thank Tudor Girba and Neil Ernst for their feedback and ideas.

7.2 Labelling Developer Topics

Within the field of mining software repositories, many approaches, such as topic modeling and concept location, rely on the automated application of machine learning algorithms to corpora. Unfortunately the output of these tools is often difficult to distinguish and interpret as they are often so abstract. Thus to have a meaningful discussion about the topics of software development, we must be able to devise appropriate labels for extracted topics.

However, these approaches neither use domain-specific knowledge to improve results, nor contextualize those results for developers. While too much specificity can produce non-generalizable results, too little produces broad learners that do not provide much immediately useful detail.

This chapter implements *labelled topic extraction*, in which topics are extracted from commit comments and given labels relating to a cross-project, software specific, taxonomy. We focus on non-functional requirements related to software quality as a potential generalization, since there is some shared belief that these qualities apply broadly across many software systems and their development artifacts.

We evaluated our approach with an experimental study on two large-scale database projects, MySQL and MaxDB. We extracted topics using Latent Dirichlet Allocation (LDA) from the commit log comments of their source control systems (CVS and BitKeeper). Our results were generalizable across the two projects, showing that non-functional requirements were commonly discussed, and we identified topic trends over time. Our labelled topic extraction technique allowed us to devise appropriate, context-sensitive labels across these two projects, providing insight into software development activities [2].

7.2.1 What's in a Name?

What's in a name? that which we call a rose
By any other name would smell as sweet;

– Romeo and Juliet, II:ii

[2] This section is based upon a currently unpublished paper, "What's in a name? On the automated topic naming of software maintenance activities" by Abram Hindle, Neil A. Ernst (a fellow grad student), Michael W. Godfrey and Richard C. Holt and John Mylopoulos [71]

Few would argue that software development is a bed of roses. To most of us, the important properties of a rose are sensual in nature, and concern appearance, odour, and feel. While horticulturalists and botanists may see more semantic depth than this, one rose is often just as pleasing to us as another. Software development artifacts, on the other hand, are abstract and intangible. They are often unnamed and ephemeral — even more so than roses — and partially derived from other unnamed and ephemeral artifacts. Yet to have meaningful discussions about how software development is progressing, we must be able to devise appropriate labels for our development topics and be able to categorize the development artifacts as belonging to one or more of these topics.

Topics arise from the wide variety of issues which occur during a software project's life-cycle. These topics can relate to, among others, the problem domain, the processes and tools used, or the development artifacts themselves. The set of development topics for a given project can sometimes be extracted automatically by analyzing artifacts within software repositories, such as change-log comments that developers create when committing revisions to the project's source control system. In the previous section, our previous work dealt with topic trends, which are topics that recur over time [80]. We observed that topic trends were often non-functional requirements.

Topics in this section and the previous section are word bags or word distributions. These word distributions are found via Latent Dirichlet Allocation [18], which finds independent word distributions shared among documents (change-log comments). The unfortunate aspect of topics that are word distributions is that they lack the tangibility of a rose. They do not self identify with their tangible properties. Topics have to be identified by interpreting the prevalent words in the word distribution and by inspecting related documents. This is impractical when one has to handle more than one hundred different topics. It would be nice if we could have automatic assistance to determine what the topic is about. This is the power of labelling and naming.

A topic — that is, a word distribution in our world — needs a suggestive name to succinctly convey its intent, and make it easy to use in discussions and analyses about the development of the system. Our previous experience leads us to believe that automated topic naming is currently infeasible in the general case. We have therefore decided to focus on topic labels from the sub-domain of non-functional requirements related to software quality.

Traditionally topic extraction has required manual annotation to derive domain-relevant labels. This section implements *labelled topic extraction*, in which topics are extracted and given labels relating to a cross-project taxonomy. This is an ontological approach where we attempt to relate networks of words to labels and then search for these terms within our topics. We also compare this approach to machine learners.

Our contributions, in this chapter, include:

- We introduce labelled topic extraction, and demonstrate its usefulness compared to other approaches.
- We show that these labels with their topics can be learnt and used to classify other data-sets.
- We present visualizations of named topics and their trends over time to aid communication and analysis.
- We use an exploratory case study of multiple open source database systems to show how named topics can be compared between projects.

We first introduce some important terminology for our work. We then describe our methodology, including our data-sets, then highlight our results. We conclude with a look at related work and possible improvements.

7.2.2 Background

We provide a brief overview of software repository mining and information retrieval. This work is related to the mining software repositories (MSR) [88] research community as it deals expressly with analyzing a project's source control repository, and the messages associated with revisions therein.

Definitions

We rely on the terms defined earlier in this chapter in Section 7.1.2. A *label* is part of a title we attach to a topic, whether manually or automatically.

Area of ROC Curve is the area under the Receiver Operating Characteristic (ROC) curve, sometimes referred to as AUC [154]. ROC values reflect a score, similar to school letter-grades (A is 0.9, C is 0.6), that indicate how well a particular learner performed for the given data. A ROC result of 0.5 would be equivalent to a random learner (randomly classifying data). ROC maps to the more familiar concepts of precision/sensitivity and recall/specificity: it plots the true positive rate (sensitivity) versus the false positive rate ($1 - specificity$). A perfect learner has a ROC value of 1.0, reflecting perfect recall and precision. A ROC value of 1.0 suggests that it has only true positives and no false positives. ROC is a well accepted performance metric for the effectiveness of classification.

Topic and Concept Extraction

Topic extraction, sometimes called concept extraction, uses tools such as Latent Dirichlet Allocation (LDA) [18] and Latent Semantic Indexing (LSI) to extract independent word distributions (topics) from documents (commit log comments). By independent we mean that the word distribution is uncorrelated with other distributions, akin to statistical independence. Many researchers [109, 128, 106, 103] have applied tools like LSI and LDA to mining software repositories, in particular for analyzing source code, bugs or developer comments.

Typically a topic analysis tool like LDA will try to find N independent word distributions found within the word distributions of all the messages. Linear combinations of these N word distributions are then meant to be able to recreate the word distributions of all of the underlying messages. These N word distributions effectively form topics: cross cutting collections of words relevant to one or more documents. Our problem is that these topics are not easy to interpret, as the underlying pattern is not clear. We feel that automatic labelling or naming of these topics would be helpful with respect to interpreting the subject of a topic. LDA extracts topics in an unsupervised manner; the algorithm relies solely on the source data with no human intervention.

In topic analysis a single document, such as a commit message, can be related to multiple topics. Representing documents as a mixture of topics maps well to source code repository commits, which often have more than one purpose [80]. A topic represents both a word distribution and a group of commit log comments that are related to each other by their content. In this section a topic is a set of tokens extracted from commit messages found within a project's source control system (SCS).

One issue that arises with use of unsupervised techniques is how to label the topics. While the topic models themselves are generated automatically, what to make of them is less clear. For example, in the previous section and in our previous work [80], as well as in Baldi et al. [8], topics are named manually: human experts read the highest-frequency members of a topic and assign a keyword accordingly. E.g., for the word list *"listener change remove add fire"*, Baldi et al. assign the keyword *event-handling*. The labels are reasonable enough, but still require an expert in the field to determine them. Our technique is automated, because we match keywords from WordNet [38], a freely available graph of word relations, to words in the topic model.

Supervised learning

While unsupervised techniques (LSI and LDA are both unsupervised) are appealing in their lack of human intervention, and thus lower effort, supervised learners have the advantage of domain knowledge which typically means improved results. In supervised learning, the data is divided into slices. One slice is manually annotated by the domain expert, and the classifications he/she determines are applied to the remaining slices. In this section, we employ the WEKA [56] and Mulan [154] machine learning frameworks in order to test supervised learning.

7.2.3 Methodology

To evaluate our approach, we sought candidate systems that were mature projects and had openly accessible source control repositories. We also decided to select systems from the same application domain, as we felt the functional requirements would probably be broadly similar. We selected MySQL and MaxDB as they were open-source, partially-commercial database systems. MaxDB started in the late 1970s as a research project, and was later acquired by SAP. As of version 7.500, released April 2007, the project has 940 thousand lines of C source code [3]. The MySQL project started in 1994 but MySQL 3.23 was released in early 2001. MySQL contains 320 thousand lines of C and C++ source code.

Generating the data

For each project, we used source control commit comments, the messages that programmers write when they commit revisions to a source control repository. We leveraged the data that we gathered in [80] for this work. An example of a typical commit message is: *"history annotate diffs bug fixed (if mysql_real_connect() failed there were two pointers to malloc'ed strings, with memory corruption on free(), of course)"*. We extracted these messages and indexed them by creation time. We summarized each message as a word distribution but removed stop-words such as common English words like *the* and *at*.

From that data-set, we created an XML file which separated commits into monthly windows. This period size is smaller than the time between minor releases but large enough for there to be sufficient commits to analyze. We applied Blei's LDA implementation [18] against the word distributions of these commits, and generated lists of topics per period. We arbitrarily set the number of topics to generate to 20, because past experimentation showed that fewer topics might aggregate multiple unique topics while any more topics seemed to dilute the results and create indistinct topics. As well, more than 20 topics quickly became infeasible for inspection and it was difficult to discern the difference in topics.

[3] generated using David A. Wheeler's *SLOCCount*.

Associating labels

Topics are word distributions: essentially lists of words ranked by frequency, which can be burdensome to interpret and hard to distinguish and understand. Once we had topics for each period, we tried to associate them with a label from a list of keywords (the word-lists depend on the particular experiment) and related terms. We performed simple string matching between these topics and our lists, 'naming' a topic if it contained that word or words. We used several different word lists for comparison.

Our first word list set, exp1, was generated using the ontology described in Kayed et al. [91]. That research constructs an ontology for software quality measurement using eighty source documents, including research papers and international standards. The labels we used:

> integrity, security, interoperability, testability, maintainability, traceability, accuracy, modifiability, understandability, availability, modularity, usability, correctness, performance, verifiability, efficiency, portability, flexibility, reliability.

Our second word list set, exp2, relied on the ISO quality model (ISO9126) [85]. ISO9126 describes six high-level quality requirements (listed in Table 7.2). ISO9126 is "an international standard and thus provides an internationally accepted terminology for software quality [19, p. 58]," that is sufficient for the purposes of this research. However, the terms extracted from ISO9126 may not capture all words associated with the labels. For example, the term "redundancy" is a term that most would agree is relevant to discussion of reliability, but is not in the standard. We therefore took the words from the taxonomy and expanded them.

To construct these expanded word-lists, we used WordNet [38], an English-language "lexical database" that contains semantic relations between words, including meronymy and synonymy. We then added Boehm's 1976 software quality model [21], and classified his eleven "ilities" into their respective ISO9126 qualities. We did the same for the quality model produced by McCall et al. [112]. Finally, we analyzed two mailing lists from the KDE project to enhance the specificity of the sets. We selected KDE-Usability, which focuses on usability discussions for KDE as a whole; and KDE-Konqueror, a mailing list about a long-lived web browser project. For each high-level quality in ISO9126, we first searched for our existing labels; we then randomly sampled twenty-five mail messages that were relevant to that quality, and selected co-occurring terms relevant to that quality. For example, we added the term "performance" to the synonyms for *efficiency*, since this term occurs in most KDE mail messages that discuss efficiency.

For the third – exp3 – list of quality labels, we extended the list from exp2 using related and similar words found in WordNet. Similarity in WordNet means siblings in a hypernym tree. We do not include these words here for space considerations (but see the Appendix (Section 7.2.7) for our data repository). It is not clear the words associated with our labels are specific enough, however: for example, the label *maintainability* is associated with words *ease* and *ownership.*

Supervised learning

In order to validate how effective these word-bag approaches to topic labelling would be we had to manually make a data set to test against. For MySQL 3.23 and MaxDB 7.500, we manually annotated each extracted topic in each period with the same quality labels as exp2 (software qualities). We looked at each period's topics, and assessed what the data – consisting of the frequency-weighted word lists and messages – suggested was the topic for that period. We were able to pinpoint the appropriate label using auxiliary information as well, such as the actual revisions and files that were related to this topic. For example, for the MaxDB topic consisting of a message "exit() only used in non NPTL LINUX Versions", we tagged that topic *portability.* We compared against this data-set, but we also used it for our supervised learning based topic classification.

We first compared our previous analysis using label matching to our manual classifications to get an error rate for that process described below in Section 7.2.4.

For supervised learning, we used a suite of supervised classifiers from WEKA [56]. WEKA contains a suite of machine learning tools such as support vector machines and Bayes nets. We also used the multi-labelling add-on for WEKA, Mulan [154][4]. Traditional classifiers map our topics to a single class, whereas Mulan allows for a mixture of classes per topic, which maps to what we observed while manually labelling topics.

To assess the performance of the supervised learners, we did a 10-fold cross-validation. This is when a set is partitioned into 10 partitions and then each partition is used once as a test set and 9 other times as part of the training set of 9 partitions. We have reported these results below in Section 7.2.4.

Finally, using this data, we evaluated two research questions (see Section 7.2.4):

1. Do label frequencies change over time? That is, is a certain quality of more interest at one point in the life-cycle than some other?
2. Do the different projects differ in their relative topic interest? That is, is a particular quality more important to one project than the other projects?

[4]`http://mlkd.csd.auth.gr/multilabel.html`

Label	**Related terms**
Maintainability	testability changeability analyzability stability maintain maintainable modularity modifiability understandability + interdependent dependency encapsulation decentralized modular
Functionality	security compliance accuracy interoperability suitability functional practicality functionality + compliant exploit certificate secured buffer overflow policy malicious trustworthy vulnerable vulnerability accurate secure vulnerability correctness accuracy
Portability	conformance adaptability replaceability installability portable movableness movability portability + specification migration standardized l10n localization i18n internationalization documentation interoperability transferability
Efficiency	resource behaviour time behaviour efficient efficiency + performance profiled optimize sluggish factor penalty slower faster slow fast optimization
Usability	operability understandability learnability useable usable serviceable usefulness utility useableness usableness serviceableness serviceability usability + gui accessibility menu configure convention standard feature focus ui mouse icons ugly dialog guidelines click default human convention friendly user screen interface flexibility
Reliability	fault tolerance recoverability maturity reliable dependable responsibleness responsibility reliableness reliability dependableness dependability + resilience integrity stability stable crash bug fails redundancy error failure

Table 7.2: NFRs and associated word-list words used in `exp2`. These words were used to label a topic with a related NFR

Measure	exp1	exp2	exp3
Topics	420	420	420
Named topics	281	125	328
Unnamed topics	139	295	92

Table 7.3: Automatic topic labelling for MaxDB 7.500. This table shows how many topics from MaxDB 7.500 were named by the word-list topic-labeller and how many were not for experiments exp1, exp2, and exp3.

7.2.4 Observations and evaluation

In this section we discuss the data and the evaluation of the experiments such as *exp2* and *exp3*.

Word list similarity

In general, word list similarity approach did not work out well as common labels dominated the less common labels. The related words for *correctness*, for example, tended to be too lengthy and non-specific. Table 7.3 lists results. A *named topic* is a topic with a matching label. We told LDA to extract 20 topics per period. All experiments were run on MaxDB 7.500 and MySQL 3.23 data.

For exp1, our best performing and most frequent labels, were *correctness* (182 topics) and *testability* (121). We did not get good results for usability or accuracy, which were associated with fewer than ten topics. We also looked for correlation between our labels: Excluding double matches (self-correlation), our highest co-occurring terms were verifiability and traceability, and testability and correctness (76 and 62 matches, respectively).

For exp2, there are many more unnamed topics. Only reliability produced a lot of matches, mostly with the word 'error'. Co-occurrence results were poor.

For exp3, we got many more named topics. As we mentioned, the word-lists are quite broad, so there are likely to be false-positives. See the following sections for our error analysis. We found a high of 265 topics for usability, with a low of 44 topics for maintainability. Common co-occurrences were reliability and usability, efficiency and reliability, and efficiency and usability (200, 190, and 150 topics in common, respectively).

Analysis of the unsupervised labelling

Based on the labels, and our manual topic labelling, we compared the results of the unsupervised word matching approach. For each quality we tried to assess whether the manual tag matched the unsupervised label assigned. Table 7.4 shows our results for MaxDB and MySQL. In general results are poor. Using the F-Measure, the weighted average of precision and recall, where 1 is perfect, our best results are 0.6, a few at 0.4, and most around 0.2. We achieved similar results using the Matthew's correlation coefficient (used to measure efficacy where classes are of different sizes) and ROC.

Based on these results we find that reliability and usability worked well for MaxDB in exp2 and better in exp3. MySQL had reasonable results within exp2 for reliability and efficiency. MySQL's results for efficiency did not improve in exp3 but other qualities such as functionality did improve. If a C grade performance has a ROC value of 0.6 then most of these tests scored a grade of C or less, but our results were still better than random chance.

Analysis of the supervised labelling

We took our annotated data-set and applied supervised learners to it. Because our data-set was of word counts we expected Bayesian techniques, often used in spam filtering, to perform well. We also tried other learners that WEKA [56] provides: rule learners, tree learners, vector space learners, and support vector machines. Table 7.5 shows the performance of the best performing learner per label. We considered the best learner for a label to be the one which had the highest ROC value for that label. Table 7.5 uses the ZeroR learner as a baseline, since it naively chooses the largest category all of the time. The ZeroR difference is often negative. For labels which are not as common, this can be expected because any miscategorization will hurt accuracy. This is why the F1 (F-measure) and the ROC values are useful, as they can better present performance on labels which are not applicable to the majority of samples.

Table 7.5 shows that MaxDB and MySQL have quite different results, as the ROC values for reliability and functionality seem swapped between projects. It should be noted that for both projects Bayesian techniques did the best out of a wide variety of machine learners tested. Discriminative Multinomial Naive Bayes (DMNBtext), Naive Bayes (NaiveBayes) and Multinomial Naive Bayes (NaiveBayesMultinomal) are all based on Bayes's theorem and all assume, naively, that the features are independent. The features we used are word counts per message. One beneficial aspect of this result is that it suggests we can have very fast training and classifying since Naive Bayes can be calculated in $O(N)$ for N features.

The smaller the label the harder it is to get accurate results. Nevertheless, these results are better than our previous word bag results of exp2 and exp3, because the ROC values are sufficiently higher in most cases (other than MaxDB reliability and MySQL efficiency).

The limitation of the approach we took here is that we assume labels are independent; however, labels could be correlated with each other. We also did not evaluate how well the learners performed together.

Experiment	Label	F1	MCC	Precision	Recall	ROC
MaxDB exp2	portability	0.228	0.182	0.520	0.146	0.553
	efficiency	0.217	0.125	0.237	0.200	0.558
	reliability	0.380	0.340	0.246	0.829	0.765
	functionality	0.095	0.083	0.250	0.059	0.521
	maintainability	0.092	0.123	0.571	0.050	0.520
	usability	0.175	0.138	0.113	0.389	0.620
	total	0.236	0.127	0.248	0.225	0.561
MySQL exp2	portability	0.138	0.211	1.000	0.074	0.537
	efficiency	0.345	0.327	0.476	0.270	0.625
	reliability	0.425	0.287	0.348	0.545	0.669
	functionality	0.025	0.006	0.571	0.013	0.501
	maintainability	0.000	0.000	0.000	0.000	0.500
	usability	0.175	0.135	0.200	0.156	0.560
	total	0.167	0.095	0.403	0.105	0.527
MaxDB exp3	portability	0.472	0.286	0.402	0.573	0.660
	efficiency	0.223	0.068	0.130	0.778	0.549
	reliability	0.257	0.196	0.149	0.927	0.652
	functionality	0.236	0.187	0.138	0.824	0.665
	maintainability	0.338	0.112	0.266	0.463	0.566
	usability	0.108	0.094	0.057	0.944	0.595
	total	0.258	0.093	0.160	0.671	0.568
MySQL exp3	portability	0.413	0.170	0.564	0.325	0.574
	efficiency	0.158	0.105	0.089	0.703	0.608
	reliability	0.388	0.240	0.260	0.758	0.660
	functionality	0.652	0.240	0.655	0.649	0.620
	maintainability	0.203	0.007	0.240	0.175	0.503
	usability	0.105	0.013	0.057	0.688	0.513
	total	0.362	0.076	0.284	0.499	0.544

Table 7.4: Results for automatic topic labelling. F1 = f-measure, MCC = Matthew's correlation coeff., ROC = Area under ROC curve

Label	Project	Learner	ROC	ZeroR Acc.	ZeroR. Diff	F1
portability	MySQL	NaiveBayesMultinomial	0.74	58.53	11.43	0.62
efficiency	MySQL	NaiveBayes	0.67	93.69	-7.68	0.23
reliability	MySQL	NaiveBayes	0.73	83.11	-12.80	0.41
functionality	MySQL	DMNBtext	0.81	54.44	21.67	0.77
maintainability	MySQL	DMNBtext	0.78	76.62	3.41	0.32
usability	MySQL	NaiveBayes	0.75	94.54	-5.80	0.21
portability	MaxDB	NaiveBayes	0.84	77.12	2.06	0.61
efficiency	MaxDB	NaiveBayes	0.62	88.43	-11.31	0.25
reliability	MaxDB	DMNBtext	0.84	89.46	3.86	0.57
functionality	MaxDB	NaiveBayes	0.67	91.26	-6.94	0.31
maintainability	MaxDB	NaiveBayes	0.70	79.43	-9.25	0.42
usability	MaxDB	NaiveBayes	0.56	95.37	-4.37	0.00

Table 7.5: Per label, per project, the best learner for that label. ROC value rates learner performance, compared with the ZeroR learner (a learner which just chooses the largest category all of the time). F1 is the F-measure for that particular learner.

Applying multiple labels to topics

We applied the Mulan [154] library for Multi-Label learning to our data-set because intuitively, topics can have more than one label, much like how a particular source code revision can have more than one topic. Multi-label learning is more than just classifying entities with more than one label. It also includes methods for determining the performance of such techniques. The problem framed in the learners above has changed; instead of looking at the precision and recall of applying one label, we rank multiple labels at once. We must check if the full subset of labels was applied, and then how much of that subset was applied.

Another aspect of multi-label learning are micro versus macro measurements. Macro measurements are aggregated at a class or label level. Micro measurements are aggregated at a decision level. So a macro-ROC measurement is the average ROC over the ROC values for all labels, where a micro-ROC is the average ROC over all examples that were classified. Unfortunately for MaxDB, the macro-ROC values are undefined because of poor performance of one of the labels.

We have presented the results of Mulan's multi-label learners in Table 7.6. Binary Relevance (BR), Calibrated Label Ranking (CLR) and Hierarchy Of Multi-label classifiERs (HOMER), performed the best. HOMER and BR act as a hierarchy of learners: BR is flat, while HOMER tries to build a deeper hierarchy to build a more accurate learner [154]. These classifiers performed better than other multi-label classifiers. They have the best

Performance Metric	MySQL BR	MySQL CLR	MySQL HOMER	MaxDB BR	MaxDB CLR	MaxDB HOMER
example Accuracy	0.45	0.40	0.50	0.49	0.47	0.51
example F1	0.56	0.48	0.59	0.55	0.50	0.53
example Hamming Loss	0.24	0.19	0.18	0.19	0.14	0.14
example Precision	0.50	0.51	0.60	0.50	0.49	0.53
example Recall	0.65	0.46	0.58	0.61	0.50	0.53
example Subset Accuracy	0.19	0.24	0.31	0.40	0.43	0.45
label macro-ROC	0.74	0.66	0.64	NaN	NaN	NaN
label macro-F1	0.46	0.30	0.43	0.32	0.19	0.23
label macro-Precision	0.41	0.41	0.49	0.29	0.31	0.32
label macro-Recall	0.56	0.26	0.40	0.38	0.15	0.20
label micro-ROC	0.81	0.81	0.77	0.76	0.66	0.62
label micro-F1	0.59	0.53	0.61	0.39	0.27	0.34
label micro-Precision	0.51	0.69	0.66	0.34	0.42	0.49
label micro-Recall	0.68	0.43	0.56	0.47	0.21	0.26
rank Avg. Precision	0.76	0.77	0.69	0.54	0.57	0.54
rank Coverage	1.51	1.47	1.94	1.40	1.23	1.43
rank One-error	0.40	0.39	0.47	0.81	0.79	0.81
rank Ranking Loss	0.19	0.18	0.27	0.41	0.35	0.41

Table 7.6: MySQL and MaxDB Mulan results per Multi-Label learner

micro and macro ROC scores, although their results seem comparable to the naive Bayesian learners we used in Section 7.2.4.

Summary of techniques

Very rarely did `exp2` and `exp3` (naive word matching) ever perform as well as the machine learners. For MaxDB, reliability was slightly better detected using the static word list of `exp2`. In general, the machine learners and `exp3` did better than `exp2` for both MaxDB and MySQL. For both MySQL and MaxDB usability was better served by `exp2`. However, usability was a very infrequent label.

We found that the multi-label learners of BR, CLR and HOMER did not do as well for Macro-ROC and Micro-F1 as NaiveBayes and other NaiveBayes derived learners did. This suggests that by sticking together multiple NaiveBayes learners we could probably label sets of topics effectively, but it would require a separate NaiveBayes learner per label.

Visualization

We have created two visualizations of the manually labelled data. A problem we faced while visualizing was how to display tag overlaps. Our solution, while not optimal, was to assign a separate colour to each distinct set of annotations. Figures 7.8 and 7.9 show extracted topics grouped by annotations. Annotations or labels that occur in adjacent windows are joined together as trends. For MaxDB (Figure 7.8) the largest trends were related to maintainability and portability. MaxDB supports numerous platforms and portability was a constant issue facing the project's development. MySQL was different, with many topics overlapping, so there were many different subsets. Functionality trends were prevalent throughout its history (the largest reddish streak across the top). Two combination tags of "functionality portability" (orange) and "maintainability portability" (teal) streaked across most of the history of MySQL. Since this was MySQL 3.23, a stable branch, we expect that issues dealing with portability and maintenance would be the primary concerns of this branch: developers probably wanted it to work with current systems and thus had to update it.

Comparing MaxDB and MySQL

We observed that MySQL had more topic subsets than MaxDB. MySQL 3.23 also had more topics and a longer recorded history than MaxDB 7.500. We tagged more MySQL topics with annotations than MaxDB topics yet both shared similarities. In terms of non-functional requirements both projects had long running trends that focused on functionality, maintainability, and portability, yet MaxDB had more of a focus on efficiency and reliability. MaxDB differed from MySQL since MaxDB was being actively developed, whereas halfway through our history of MySQL 3.23, other versions of MySQL were being actively developed: MySQL versions 4.0, 4.1, and eventually 5.0 and 5.1. In other words, MySQL 3.23 was being maintained rather than actively developed, whereas MaxDB 7.500 was being actively developed into MaxDB 7.6, and then maintained thereafter.

MySQL and MaxDB's machine learners did make decisions based off some shared words. Words that were used to classify topics that were shared between MySQL and MaxDB included: bug, code, compiler, database, HP UX, delete, memory, missing, problems, removed, add, added, changed, problem, and test. Adding these words to the word bags of exp2 and exp3 could improve performance while ensuring they were only domain specific.

With respect to the questions raised in Section 7.2.3:

Do label frequencies change over time? – Yes, MySQL's label frequencies decreased as it got older. Usability and reliability labels became more and more infrequent as it matured. Maintainability topics became more prevalent as MaxDB matured.

Do the different projects differ in their relative topic interest? – Yes. MySQL 3.23 had proportionally more functionality labelled topics, while MaxDB had proportionally more efficiency and portability related topics.

Annotation observations

We found many topics that were not non-functional requirements (NFRs) but were often related to them. For instance, concurrency was mentioned often in the commit logs and was related to correctness and reliability, because concurrency was troublesome. Configuration management and source control related changes appeared often and sometimes there were topics dedicated to configuration management. These kinds of changes are slightly related to maintainability. A non-functional change that was not quality-related was licensing and copyright; many changes were simply to do with updating copyrights or ensuring copyright or license headers were applied to files.

We noticed that occasionally the names of modules would conflict with words related to other non-functional requirements. For instance, optimizers are very common modules in database systems: both MySQL and MaxDB have optimizer modules. In MySQL the optimizer is mentioned but often the change deals with correctness or another quality. Despite this difference, the name of the module could fool our learners into believing the change was always about efficiency. Perhaps a project specific word-bag is needed in order to avoid automated mistakes due to the names of entities and modules of a software project.

Effectiveness

With ROC values ranging from 0.6 to 0.8 we can see there is promise in these attempts. `exp2` and `exp3` both indicate that static information can be used to help label topics without any training whatsoever. If the techniques used in `exp2` and `exp3` were combined with the supervised techniques we could probably reduce the training effort. Both Naive Bayesian learners and the word-list approaches were computationally efficient. These results are promising, because the results are accurate enough to be useful, while still cheap enough to execute to be feasible as an automated or semi-automated method of labelling topics by their software qualities.

Threats to validity

Our study suffers from multiple threats to validity. *Construct validity* issues include that we used only commit messages rather than mail or bug tracker messages. We also chose our taxonomy and the data to study. *Internal validity* issues are to do with inter-rater reliability. *External validity* issues are that our data originated from FLOSS database projects and thus might not be applicable to commercially developed software. Furthermore, our analysis techniques rely on a project's use of meaningful commit messages.

7.2.5 Related work

The idea of extracting higher-level 'concerns' (also known as 'concepts', 'aspects' or 'requirements') has been approached in two ways.

Cleland-Huang and her colleagues published work on mining requirements documents for non-functional requirements (NFR) (quality requirements) [26]. One approach they tried was similar to this one, with keywords mined from NFR catalogues found in their previous work [25]. They managed recall of 80% with precision of 57% for the Security NFR, but could not find a reliable source of keywords for other NFRs. Instead, they developed a supervised classifier by using human experts to identify an NFR training set. There are several reasons we did not follow this route. One, we believe we have a more comprehensive set of terms due to the taxonomy we chose. Secondly, we wanted to compare across projects. Their technique was not compared across different projects and the applicability of the training set to different corpora is unclear. A common taxonomy allows us to make inter-project comparison (subject to the assumption that all projects conceive of these terms in the same way). Thirdly, while the objective of Cleland-Huang's study was to identify new NFRs (for system development) our study assumes these NFRs are latent in the textual documents of the project. Finally, the source text we use is less structured than their requirements documents.

In the same vein, Mockus and Votta [120] studied a large-scale industrial change-tracking system. They also leveraged WordNet, but only for word roots. They felt the synonyms would be non-specific and cause errors. A nice contribution was access to system developers, with whom they could validate their labels. Since we try to bridge different organizations, these interviews are infeasible (particularly in the distributed world of open-source software).

The other approach is to start with code repositories, and try to extract concerns from there. Marcus et al. [109] describe their use of Latent Semantic Indexing to identify commonly occurring concerns for software maintenance. Some results were interesting, but their precision was quite low. ConcernLines [152] shows tag occurrence using colour intensity. They mined change request tags (such as 'milestone 3') and used these to make evolutionary analyses of a single product. The presence of a well-maintained set of tags is obviously essential to the success of this technique.

Mens et al. [117] conducted an empirical study of Eclipse, the open source software (OSS) source code editor, to verify the claims of Lehman [100]. They concerned themselves with source code only, and found Law Seven, "Declining Quality", to be too difficult to assess: “[we lacked an] appropriate measurement of the evolving quality of the system as perceived by the users [117, p. 388]”. This chapter examines the notions of quality in terms of a consistent ontology, as Mens et al. call for in their conclusions.

Mei et al. [113] use context information to automatically name topics. They describe probabilistic labelling, using the frequency distribution of words in a topic to create a meaningful phrase. They do not use external domain-specific information as we do.

In [37], Neil Ernst describes an earlier project that this work was based on, to identify change in quality requirements in GNOME software projects; this approach is solely text-matching, however, it does not leverage machine learning strategies.

7.2.6 Conclusions and future work

We demonstrated that static but domain-specific knowledge can improve unsupervised labelling of extracted topics. Our `exp2` experiment used small accurate word bags to label topics but performed just as well as `exp3`, which used many more general terms from WordNet. We then showed that with some supervision, and by using efficient machine learners based on Naive Bayesian classifiers, we could improve the accuracy of automatic labelling topics even further.

Our manual inspection and annotation of the topics extracted from MySQL and MaxDB revealed that many of the extracted topics dealt with non-functional requirements, and these topics were spread across the entire history of a project. In the cases of MaxDB and MySQL, portability was a constant maintenance concern and was prevalent throughout the entire lifetime of the projects.

We showed that non-functional requirements are often trending topics, that non-functional requirements are quite common in developer topics, and that there are efficient methods of semi-automating and automating topic labelling.

There are many avenues of further investigation. We want to investigate developer attitudes related to these labels: i.e., when we label a topic, was the developer expressing positive or negative qualities about that label? It is difficult to map abstract qualities to particular messages. Is a “quality” discussion about more than just corrective maintenance?

7.2.7 Appendix

Our data and scripts are available at `http://softwareprocess.es/nomen/`

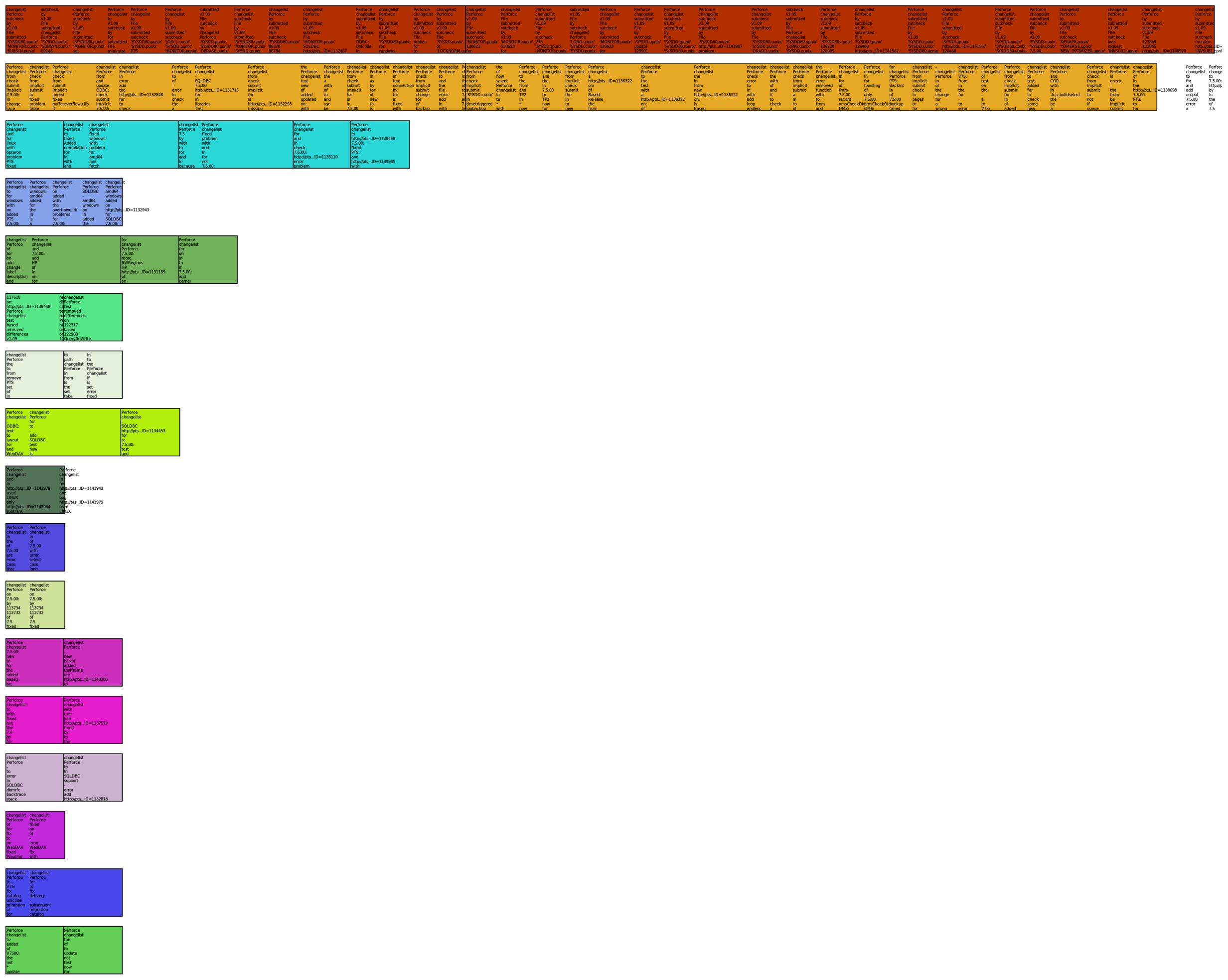


Figure 7.6: The top part of a trend-histogram of MaxDB 7.500, ordered by topic occurrence. X-axis determines the number of continuous months of a trend. Trends are ranked by the number of topics that a trend contains in descending order.

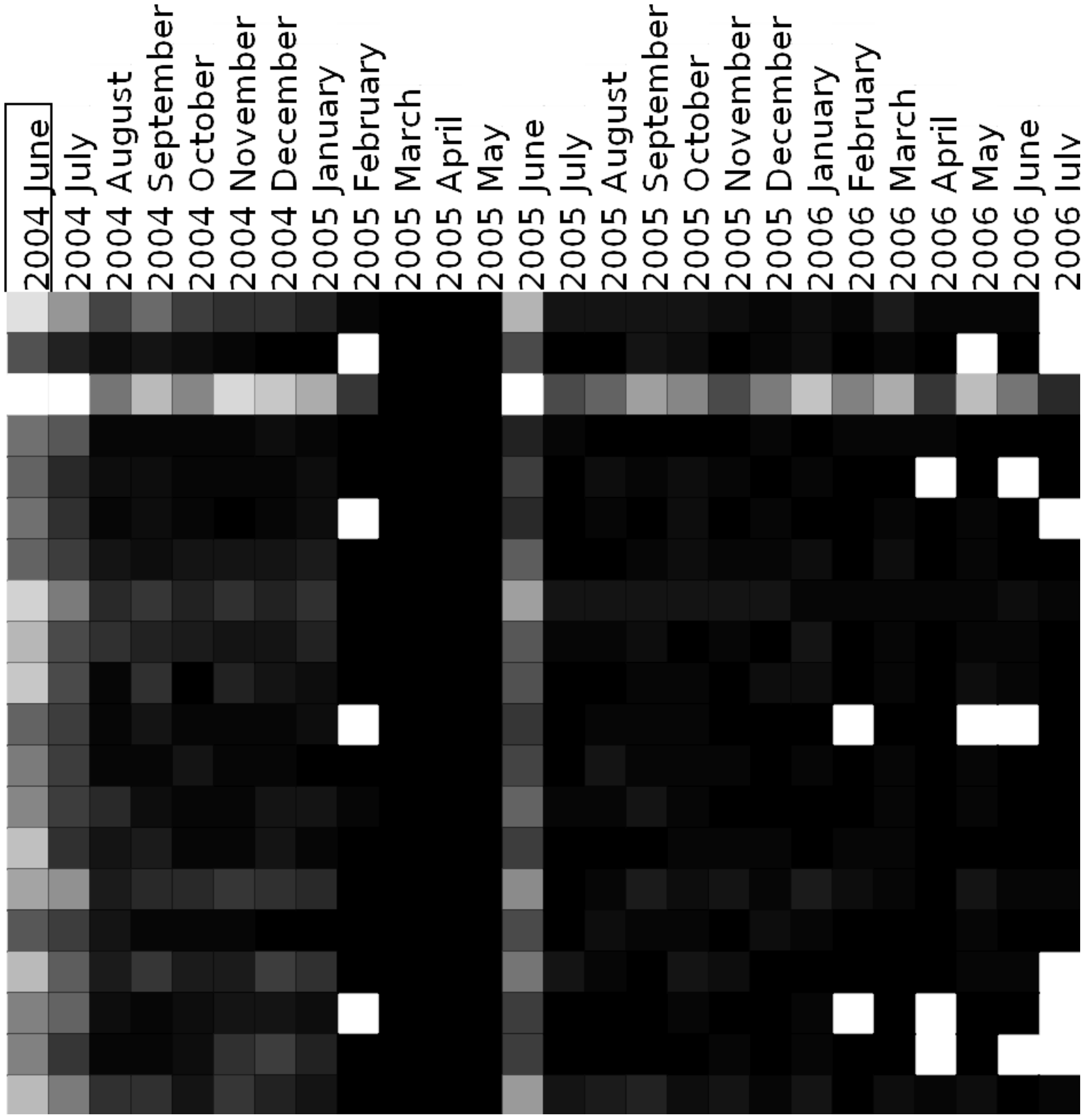


Figure 7.7: This figure depicts MaxDB 7.500 topics analyzed with global topic analysis, 20 topics (each row) and their document counts plotted over the entire development history of MaxDB 7.500 (26 months). The shade of the topic indicates the number of documents matching that topic in that month relative to the number of documents (white is most, black is least).

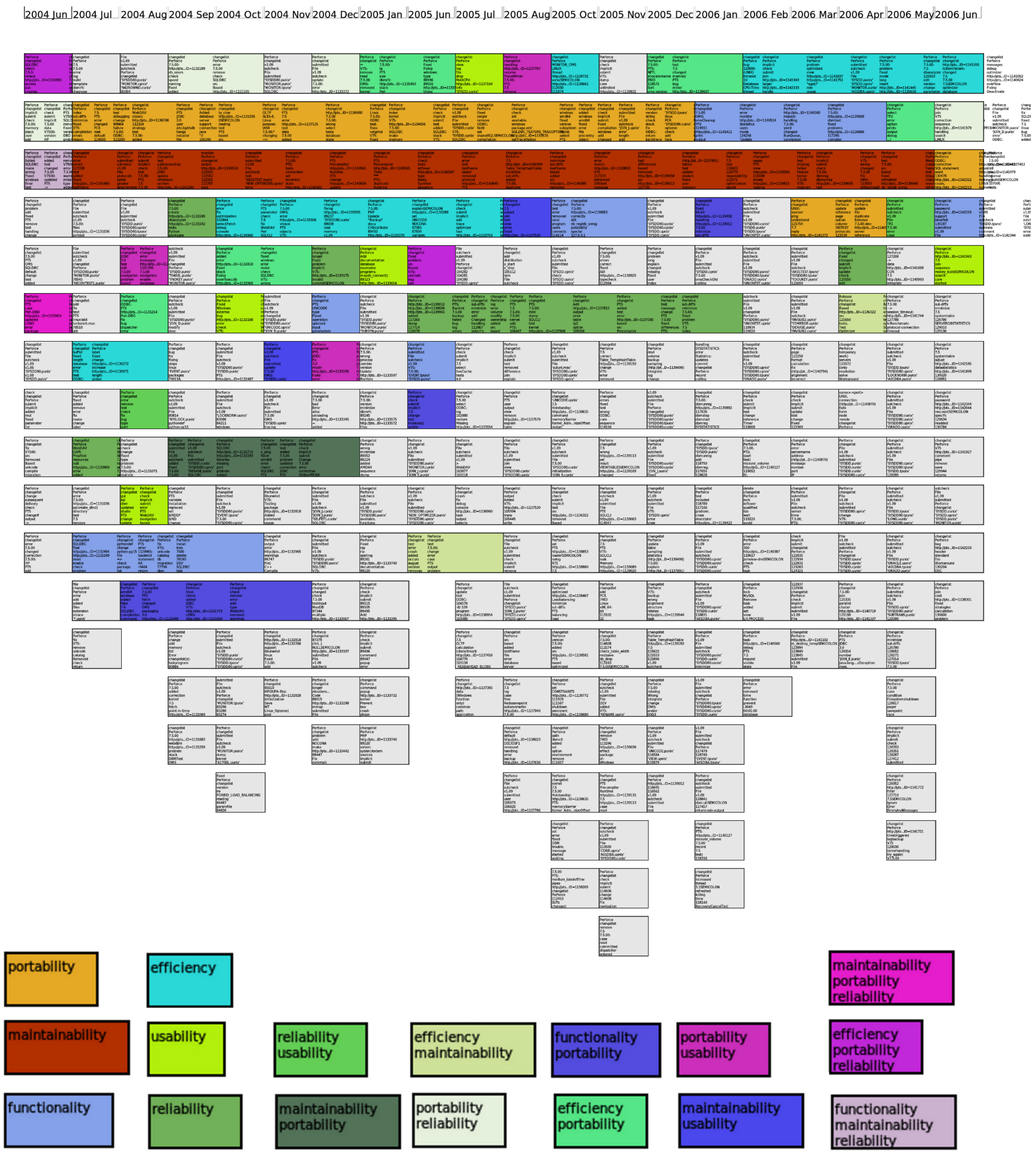


Figure 7.8: MaxDB 7.500: Labelled topics related to non-functional software qualities plotted over time, continuous topics are trends that occur across time

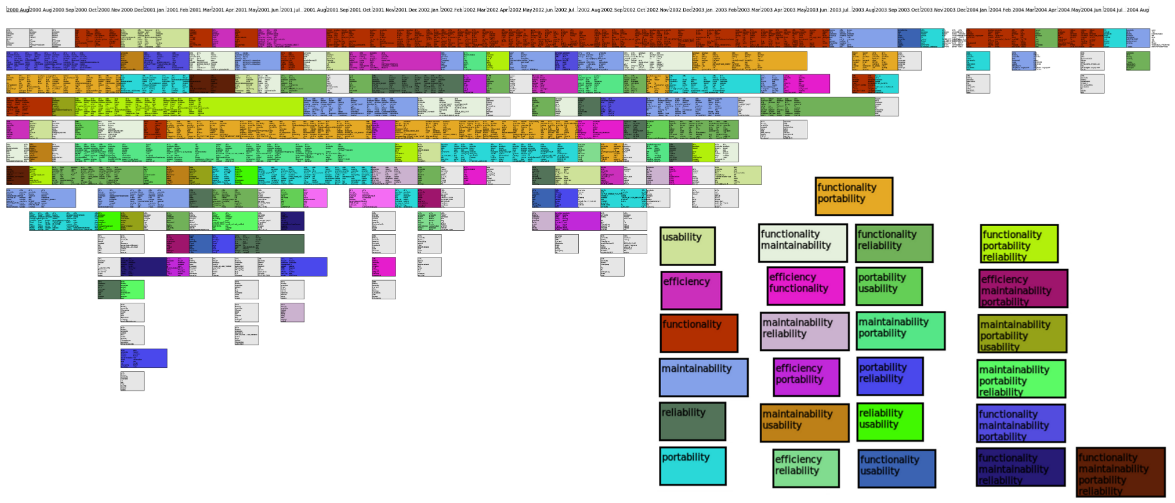


Figure 7.9: MySQL 3.23: Labelled topics related to non-functional software qualities plotted over time, continuous topics are trends that occur across time

Chapter 8

Synthesizing Recovered Unified Process Views

This chapter integrates the previous chapters to produce general process overviews. We rely on multiple repositories in this chapter: source control systems, bug trackers, and mailing list archives.

This chapter is meant to demonstrate that some processes are observable based on evidence left behind, from a global perspective. This is an attempt to provide a classical software engineering view for concrete software development processes. We produce a view of underlying processes from a Unified Process perspective. Thus what is demonstrated here and integral to this thesis is that there is evidence of process, it is observable, it changes over time, and it is composed of concurrent and parallel behaviours.

This chapter also covers the other disciplines of the Unified Process that were not covered in much detail in previous chapters. Such disciplines include requirements, design, deployment, project management and quality assurance.

The development process for a given software system is a combination of an idealized, prescribed model and a messy set of ad hoc practices. To some degree, process compliance can be enforced by supporting tools that require various steps be followed in a particular order; however, this approach is often perceived as heavyweight and inflexible by developers, who generally prefer that tools support their desired work habits rather than limit their choices. An alternative approach to monitoring process compliance is to instrument the various tools and repositories that developers use — such as source control systems, bug trackers, and mailing list archives — and to build models of the de facto development process through observation, analysis, and inference. In this chapter, we present a technique for recovering a project's software development processes from a variety of existing artifacts. We first apply unsupervised and supervised techniques — including word-bags, topic analysis, summary statistics, and Bayesian classifiers — to annotate software artifacts by related topics, maintenance types, and non-functional requirements. We map these analysis results onto a time-line based view of the Unified Process development model, which we call Recovered Unified Process Views. We demonstrate our approach for extracting these process views on two case studies: FreeBSD and SQLite [1].

8.1 Investigating Development Processes

A custom more honour'd in the breach than the observance.

– Hamlet, I:iv

In principle, software development processes offer a means to ensure reliable and predictable results when creating and modifying software systems. While there are a number of well known processes and process frameworks — and the research literature on this topic is extensive [138, 20, 144] — companies commonly follow a "home-brew" development process that mixes broad and well known process practices with narrower and more specialized ones that are dictated by the particular demands of their business model.

Anecdotal evidence suggests that even these customized process models are often not followed very closely in practice. Consequently, if managers want to track what actual process steps their developers have been following, they must either interview the developers or analyze the observable results of their efforts.

The advantages of interviewing developers are numerous: one can ask both general, wide-ranging questions as well as directed questions about specific areas of interest; one can adapt the line of enquiry as new information is revealed; and the replies are usually rich in detail, often providing additional context to the interviewer that they may have been unaware of. However, interviewing also has many disadvantages: it is labour intensive and time consuming for both the interviewer and interviewee; it is hard to automatically extract and organize the large amounts of details that typically result; developers may view the interview as adversarial, and may be unwilling to cooperate truthfully; and developers may have a mistaken view of what process steps they actually followed. Additionally, stakeholders other than team managers may wish to be able to examine the development process of a software system. For example, new team members might desire a way to learn about a project and its development culture without bothering their colleagues; also, regulating agencies and potential corporate partners may wish to be able to scrutinize

[1] This chapter is based on the paper "Software Process Recovery using Recovered Unified Process Views" published at ICSM 2010 written by Abram Hindle, Michael W. Godfrey, and Richard C. Holt [81].

development practices as part of a due diligence effort. In these cases, the stakeholders may have limited or no direct access to the developers in question, and semi-automated approaches may be the only practical option.

The advantages of *software process recovery* through semi-automated artifact analysis include that it is largely automated, it is relatively easy to gather the needed data — assuming that it is available in the first place — and that, apart from validation, it does not require access to the developers. However, there are also disadvantages: it can be hard to meaningfully link data from disparate sources; there are often large semantic gaps in the available knowledge; and even if results of one study seem promising, it is unclear how generalizable they may be.

This research explores the extent to which semi-automatic analysis of development artifacts can be used to extract the processes used by the developers. Our goal is to be able to map artifacts to process-oriented activity models such as the UP time-line diagram shown in Figure 2.1. The artifacts of interest we consider in this research include changes to source code and documentation, bug tracker reports and events, and mailing list messages. We analyze these artifacts to extract indicators of use and behaviour. We also perform signal (event-stream) similarity analysis (Section 8.4.2) to determine when behaviour within a repository significantly changes. And finally we perform a topic analysis of the artifacts; that is, we attempt to characterize the intent or purpose of the artifacts by comparing the natural language used within them against a benchmark glossary of general software engineering concepts, such as non-functional requirements. By analysis and presentation of the underlying purpose, topics, processes, and behaviour we provide a historical view of the software development processes, as illustrated in Figure 8.2.

Software process recovery and RUPVs can be applied to help stakeholders discover what processes were used to build a project. These stakeholders range from managers to investors. Software process recovery can be used to provide managers with project dashboards. New developers can use RUPVs to determine when certain disciplines are used within a project's life cycle. Companies can use software process recovery and RUPVs to help document their processes for ISO 9000 [147] compliance. Investors can use RUPVs to determine how a product was built and investigate what are the development processes used within an organization. Essentially RUPVs and software process recovery are useful to anyone with an interest in a project's development process.

The contributions of this chapter include:

- a proposal of a methodology for software process recovery,
- a proposal of a high-level process visualization called the Recovered Unified Process Views (Figure 8.2), that is based on UP time-lines (Figure 2.1), and
- a case study of FreeBSD and SQLite using these techniques.

8.1.1 Motivation

Figure 2.1 depicts the time-line of development activities of an idealized project through the lens of the Unified Process (UP). We call this kind of model an UP diagram; it is a well known and elegant visualization of the multi-dimensional nature of iterative software development. In this diagram, we can see a mixture of concurrent and parallel behaviours and workflows that span different development disciplines. Since a diagram like this is effective at communicating a prescribed process, we conjecture that it will be useful to describe the actual underlying and observed process. In our approach to semi-automatic recovery of a project's process, we create diagrams that are similar to these UP diagrams.

A persistent problem with this kind of approach is that the activities within the various UP disciplines may not be easily observable by simply monitoring available development artifacts. For example, project requirements may be managed by a group external to the development team, and key but undocumented design decisions may be made informally between developers in face-to-face meetings. These holes in the record present a challenge to meaningful analysis: if we cannot observe all activities of the ongoing development effort, how reliable are our results likely to be, and what can we do with them? One thing we can do is to use the events that we *can* observe to try to predict when undetected events might have occurred. For example, discussions about APIs and the modification of an API might be a result of the design and analysis discipline. Even then there are disciplines within the UP that may not have directly or indirectly observable events. With our approach we model what we can observe and infer, including activities that the process designers may not have considered explicitly. For example, we can focus on quality assurance (QA) and the non-functional requirements related to those qualities, such as reliability or portability. These concerns are not an explicit part of the UP, yet may be of interest to stakeholders.

8.2 Previous work

Our work leverages research primarily from the mining software repositories (MSR) community [88], which often focuses its mining efforts on source control systems, bug trackers, and mailing lists.

The focus of our work concerns software development processes. Unfortunately, the term "process" is overloaded in this research field, so we must take care to distinguish development processes from *stochastic* processes and *business* processes.[2] Stochastic processes and time-series have been used to explore the laws of software evolution [98, 51]. For example, Herraiz et al. [65] studied these processes in depth by mining many software

[2]In this chapter, unless otherwise indicated the reader should assume that the term "process" refers to a software development process.

projects and studying their software metrics; they found that metrics used to measure growth usually followed Pareto distributions.

Business processes are closer to software development processes because they concern sequences of related tasks, activities, and methods that are combined to achieve some business goal, such as the handling of product returns at a retail store. Van der Aalst et al. [156] describes business process mining as the extraction of business processes at run-time from actual business activities. Van der Aalst deferred to Cook and Wolf [29] when it came to applying process mining on software projects. Software development is a kind of information work, like research, and thus is not easy to model so formally. Heijstek et al. [63] leveraged a suite of IBM and Rational products that instrumented multiple industrial projects in order to produce UP diagrams much like our RUPVs. Their approach relies on different kinds of repositories and time sheet information.

Software development processes are meant, in an preemptive sense, to ensure quality and provide a reliable framework for the development of software projects. Some processes address a specific part of software development, such as maintenance [97]. The software development life-cycle (SDLC) [144] describes how software is often built, maintained, supported, and managed. Most software development processes relate to some if not all of the various aspects of the SDLC. Software development processes are often posed as a methodology related to development, such as the waterfall model [138] and the spiral model [20]. More recent software development processes include the Unified Process [86], (see Figure 2.1 for a diagram of the Unified Process disciplines over time), Extreme Programming (XP) [10], SCRUM [140], and many methodologies related to Agile development [66]. Most of the recent processes focus on incremental development and smaller iterations, so that design and requirements can be updated as they become clearer over time. Software development processes and life cycles seek to manage the creation and maintenance of software. Meta-processes, such as the Capability Maturity Model (CMM) [1], attempt to model the processes used to create software, much like ISO standard 9000 [147] attempts to model and document how processes are executed, tracked, modelled and documented. The CMM concerns modelling, tracking, and ranking software process adherence.

Our work on non-functional requirements [71] that we extract from software artifacts is based on Ernst et al.'s [37] work and is also related to Cleland-Huang et al.'s [26] work on mining requirements documents for non-functional requirements (NFR). Cleland-Huang used keywords mined from NFR catalogues [25]. With respect to the MSR community Mockus and Votta [120] leveraged WordNet and word-bag approaches to discover commits that dealt with security topics. Treude et al. [152] produced ConcernLines, a visualization and methodology that mines manually created tags from software in order to present views of the software history and processes.

All of this work is relevant to our attempts to produce Recovered Unified Process Views (RUPV) where we need to track changes, topics of changes, and discussions over time.

8.3 Methodology

Our purpose is to produce reports on activity within repositories and to produce diagrams like the UP diagram. Our recovered versions of a UP diagram are called Recovered Unified Process Views (RUPV). RUPVs are views in that there are many possible perspectives from which to analyze and view the software development history, behaviours, and processes of a software project.

Our overall methodology can be broken down into six steps as illustrated in Figure 8.1: acquisition, extraction, unsupervised analysis, annotation, supervised analysis, signal mapping and reporting. Our methodology flows from acquisition and extraction to supervised analysis and feeds back into unsupervised analysis or transitions into signal mapping and reporting.

Source Acquisition is the discovering and mirroring of relevant repositories of data and software artifacts. Often these repositories need to be mirrored in order to avoid affecting performance of the development environment or to ensure that the repositories are archived.

Extraction extracts data from the artifacts collected during the acquisition step. The type of data (and its meta-data) depends on the repository being extracted. In Section 8.3.2 we will discuss each kind of repository that we extracted for this chapter, but the most important information we are looking for is creation and change events (revisions) of software artifacts. We can get this kind of information from source control, mailing lists and bug trackers. Depending on the kind of analysis used, we might need partial or entire artifacts: source code analysis might require the full source code but natural language processing-style (NLP-style) analysis might only need word counts of textual data.

Unsupervised analysis is the analysis of the extracted data and events, generally without the help of the end user, without annotation. Automatic methods are used in this step. Unsupervised analysis ranges from summary statistics and modelling, to NLP-like analysis and word-bag analysis, to topic analysis.

Annotation is used to enhance the unsupervised analysis by clarifying information such as classification decisions, modifying stop words, or modifying word-bag dictionaries.

Supervised analysis includes methods that require some form of human intervention such as tuning training sets or labelling commits. We use supervised analysis to label topics and classify revisions by their maintenance categories. Supervised techniques often employ machine learning based classifiers that require annotated training sets.

Signal Mapping and Reporting takes previous analyses and presents them as consumable Recovered Unified Process Views. Signals, event streams, are combined to produce process related measures that summarize the underlying observable processes.

The rest of this section will detail each of these steps and demonstrate how they fit together.

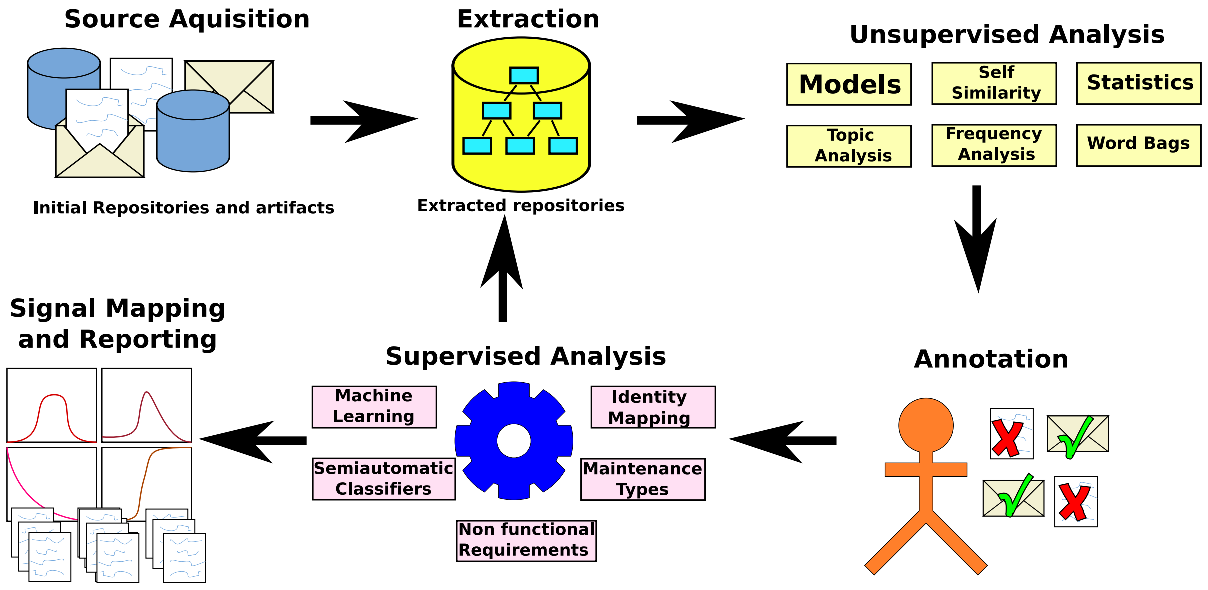


Figure 8.1: Methodology flow chart, describes how artifacts are extracted, analyzed, annotated and reported about.

8.3.1 Source Acquisition

We are interested in development events that we can extract from a wide variety of data-sources, including source code changes and bug reports. *Sources* are data-sources such as software repositories. The UP diagram illustrates how a software process might consist of parallel efforts, such as requirements and business modelling, that are related to multiple sources of data and multiple kinds of artifacts.

Depending on the project, such information might not be available. Even when artifacts, such as requirements documents, are unavailable, there still may be data that can act as its proxy. Sources are particularly important when they relate time to creation events or change events. A source can be useful even if it is difficult to analyze or assess, the simple count of an event occurring can suggest that some effort was taken with respect to a certain kind of task. These counts of events of a task could be further aggregated into larger disciplines like those Figure 2.1.

A series of events can be abstracted as a signal. *Signals* can be extracted from meetings, design document revisions, revisions to artifacts, requirements revisions, releases, milestones, deliverables, story cards and story card completions, even diagram revisions. Our two case studies in this chapter rely on public data, so we are limited to available data-sources such as source control systems, mailing lists, and bug trackers.

Ideally during acquisition we could also mine developer documents for information about mirroring or cloning the repositories of a project, such as mailing list archives or source control systems. Tools like CVSup, cvssuck, svnadmin and Git can be used to mirror source control repositories.

The acquisition step is meant to ensure that these data sources are acquirable, thus it is useful to automate this process to allow for analysis. Once acquisition of sources is complete the data must be abstracted by the extraction step before they can be analyzed further.

8.3.2 Extraction

The extraction step attempts to abstract the data from those sources collected during the acquisition step. In this step we extract events, their data, and information about when they occur. This means we have to be able to convert the raw data into a format that is usable by our various tools. This extracted data will need further analysis in order to produce usable signals. Extractors that could be useful include CREX [61], CVSAnalY [136], SoftChange [49], MLStats [136], as well as our own extraction tools. We have many kinds of repositories that we can extract various kinds of development artifacts from:

Source Control Repositories

For CVS data we use tools such as CVSup to mirror the data, and we use SoftChange and CVSanalY to extract it into a database format that allows us to easily query authors, files, revisions, and commit messages. In the case of CVS, commits are not recorded and need to be rebuilt from file revisions. To extract FOSSIL repositories, used by SQLite, we wrote our own extractor.

Mailing lists

While tools for analyzing mailing lists such as MLStats exist, we wrote our own mailing list analysis tool called MBoxTractor[3]. The data we extract from messages found in mailing lists includes: authors, direct receivers, people mentioned within the body of the message, message content, and the content without quotes. Email address normalization should also be employed to help consolidate identities.

A particular project might have multiple mailing lists dedicated to different issues such as user support, or developer discussions. These mailing lists can be combined or separated per project. A developer mailing list might be more relevant to development while traffic on a user mailing list might be more relevant to end-users of the software.

[3]Bug and email and FOSSIL extractor: `http://softwareprocess.es/MBoxTractor/`

Bug trackers

The bug tracker is a convenient source of process related information because interaction with the bug tracker is relatively formal, well recorded and annotated. Some bug trackers such as *gnats* or *FOSSIL* often are missing data of interest such as identity of the bug reporter or dates of certain events. Bug trackers are especially valuable because their bug identifiers are referenced in other repositories like the source control system. For example, the bug identifiers assigned to a bug report are often referenced on mailing lists and in repository commit messages.

Bug trackers also may serve as a source of requirements data. For example, some bug reports are marked as feature requests and are sometimes used to discuss and flesh out requirements. Bug reports are often rich with many timestamped events. We used MBoxTractor, our own bug extractor, on the reported bugs of FreeBSD and SQLite.

Traceability Links

Traceability links are meta-artifacts, that is they are references between artifacts. Traceability links highlight cross-repository aspects of the development process as information flows between repositories. An example traceability link would be a bug report number embedded in a commit log message.

People

People are the ultimate traceable artifacts within a repository as they are creators of artifacts and changes. They are referenced by artifacts across repositories as they show up in most of the repository meta-data, in the source code copyright statements, in comments, and mentions in mailing lists. If a person interacts with multiple repositories it could indicate what kind of role they play in a project. For example, someone who participates on a user mailing list may be only a user. If they are referenced within the source control system they might be a developer or they might have contributed a patch.

Unfortunately people are not generally represented as entities within a repository so they need to be identified. Even once identified many contributors use more than one email so their identities must be consolidated.

These repositories and entities discussed in this section will be used as input for a more thorough analysis in the next step: unsupervised analysis.

8.3.3 Unsupervised Analysis

The unsupervised analysis step occurs after extraction and is done automatically without the user's help. Unsupervised analysis often consists of partitioning data, decision making, and classification. This goes beyond extracting entities. The following subsections address the various kinds of supervised analysis we used in this chapter.

STBD revisions

In order to break revisions down by their purpose, we track four main file-type revisions, which we refer to as STBD [68] (addressed in Chapter 4): *source* code changes, *test* code changes, *build* system changes, and *documentation* changes. These kinds of revisions are relatively simple to track and extract with reasonable accuracy. To classify files into any of the four types we compare the filename against sets of regular expressions associated with each STBD type. STBD partitioning is useful as it allows us to break down a stream of revision events into semantically distinct streams.

Word-bags

Word-bag analysis is an unsupervised method of labelling natural language documents, such as change-log comments or bug reports, with concepts that we know of beforehand. Word-bags are word lists or dictionaries of terms related to a concept such as efficiency or requirements. If a document matches a word from the a word list we label that document with the concept of that word list, such as requirements. We reused the requirements word-bag from Chapter 7. Our *requirements* word-bag contains: *requirement, use case, stories, actors, clients, modelling, modeling, elicitation, feature, functional, quality, format, formatting, goal, definition.* While our *portability* word-bag contains: *portability, transferability, interoperability, documentation, internationalization, i18n, localization, l10n, standardized, migration, specification, portability, movability, movableness, portable, installability, replaceability, adaptability, conformance.*

Word-bag analysis is an easy-to-deploy unsupervised analysis technique that has some degree of accuracy [71]. We deploy word-bag analysis to find NFR related artifacts because NFRs are cross-project concerns that often use similar language across projects, that means once we generate a reasonable word-bag for a concept such as portability we can use that word-bag to analyze other projects.

Topic Analysis

Topic analysis [80], described in Chapter 7, tries to find independent development topics in commit log comments. Topic analysis utilizes tools such as latent Dirichlet allocation (LDA) or latent semantic indexing (LSI) to automatically find development topics. The topics produced are much like word-bags, but these are automatically extracted from the commit log messages and often represent development issues, such as bug fixing, that are addressed during development.

Unsupervised analysis can be enhanced by annotating word-bags and stop words. The next step, annotation, can employ unsupervised analysis to create training sets for supervised analysis.

8.3.4 Annotation

Human intervention is required in the annotation step as stop word lists and training-sets need to be fashioned from previous analysis steps. The annotation step occurs after the unsupervised analysis step and serves to enhance the results of the unsupervised analysis. As well annotation serves as a method to prepare training sets for supervised learners.

Stop words, such as "the" and "with", are used to eliminate common words that obscure automatic results. Sometimes common words will dominate results and it might be necessary to remove them from the data in order to enhance the results. In the annotation step one should investigate the topics generated by topic analysis and remove stop words that are not useful in distinguishing topics from each other. For instance if a source control system was automatically updated by another service, such as a vendor's source control system, the words used in the automatic update might show up in most of the topics. The removal of these words might enable more meaningful topics to be extracted.

The lexicon of a project is often unique to itself [145]. Word-Bags should be tuned appropriately by removing words that cause mismatches or adding more appropriate words. For instance, if the word *optimize* or *optimizer* refers to a module rather than the performance of a software system it might be best to avoid false classifications by removing those terms from the performance word-bags [37, 71].

Supervised analysis such as topic labelling or maintenance classification needs training sets. During the annotation step training sets can be fashioned out of previous unsupervised analyses like word-bag analysis. To use word-bag analysis to help build a training set, sample an equal number of word-bag positive examples and negative examples, and then correct these samples. This is easier than manual annotation because one does not have to manually annotate any examples that were correctly annotated by the word bag analysis. For a requirements word-bag, one could run the word-bag analysis and find and correct 50 requirements examples and 50 non-requirements examples. These training sets will be fed into the next step, supervised analysis.

8.3.5 Supervised analysis

Supervised analysis requires some human intervention, provided during the annotation step. These supervised methods can often produce better, more project relevant results than unsupervised methods. Two kinds of supervised analysis we use are topic labelling and maintenance classification.

Topics Labelled by NFRs

Topic labelling [71] attempts to label topics extracted during the unsupervised analysis step with non-functional requirements (NFRs) such as: efficiency, functionality, reliability, usability, maintainability and portability. These topics are labelled based upon the words they are composed of. We use a Bayesian classifier in order to label topics by their NFRs, thus a training set of labelled topics is needed to train the learner. By labelling topics by NFRs we can attempt to characterize some of the quality-oriented processes being followed within a software project.

Maintenance Classification

Maintenance classification tries to classify commits to the source control system by their maintenance categories based on their commit log messages [73]. The purpose is to characterize the maintenance based aspects of the underlying development process.

The maintenance categories that we use are based upon an extended version of Swanson's maintenance categories: corrective changes that fix bugs, adaptive changes dealing with the run-time environment and portability, perfective changes meant to improve maintainability or efficiency, feature additions, and non-source-code changes such as copyright statements. This is all done with a machine learning based classifier that requires a training set of commits labelled by their maintenance categories.

After our supervised analysis is complete we can either feedback into unsupervised analysis and annotation again or we can move on to aggregating and reporting our results. In this study we used one iteration of the feedback loop, as we modified stop-words per project before we produced the final reports.

8.3.6 Signal Mapping and Reporting

After completing unsupervised analysis, annotation, and supervised analysis steps we are ready to synthesize and report the results based on the previous analysis steps. Results can be summarized textually, but our focus is to produce Recovered Unified Process Views (RUPVs) that are similar to the UP diagram in Figure 2.1.

For each set of events, such as commits, bug report messages, and mailing list messages, we have signals of their events over time. We may choose to partition these signals by author interaction, by file type, or by their description.

For each set of events — commits, bug reports, and mailing list messages — we extracted word-bag derived signals for NFRs and other software engineering terms. The word-bags included efficiency, maintainability, reliability, functionality, portability, usability, requirements, analysis, deployment, and project management.

We combined signals into new signals meant to proxy or simulate UP diagram disciplines such as business modelling, requirements, analysis, implementation, testing, deployment, configuration, project management and support environment. These signals are explained in more detail in Section 8.5.

Other signals we have access to are tags from source control systems and releases. Releases are an especially valuable signal because they indicate a process driven event, the release, has occurred.

These signals and results can then be presented to the end user as described in Section 8.5. Next, in Section 8.4, we describe methods of visualizing and analyzing some of these signals.

8.4 Signals

We can collect signals from events by counting them or measuring them over time. We can deal with signals in many ways, ranging from bucketing to windowing, to moving averages, to visualization and comparison, as we will now explain.

8.4.1 Signal Visualization

Generally to produce RUPVs we want to create parallel plots of signals across time, i.e., multiple signals plotted in a stacked fashion with a common time-line (see Figure 8.2). Sometimes a signal may be too noisy to be useful so moving averages, windowing, or bucketing can be used to clean up the signal. In the cleaned up signal, we may be able to see trends in the data that would otherwise be obscured.

8.4.2 Signal Similarity

If we are analyzing process-oriented signals, we might want to see when a signal correlates with another signal or when a signal abruptly changes its behaviour. We can look for these patterns by comparing a signal to itself using self-similarity. Self-similarity is when we take periods (or regions) of the signal and compare it to other periods of the signal. We can use the same techniques to compare two signals as well.

We use two methods of signal similarity comparison, global and local. Global comparison correlates the local bins to all bins across time. The local comparison correlates the local bins to bins nearby within a certain threshold, such as an iteration or a three month window.

The comparison metric we use here is the Euclidean distance between feature vectors consisting of summary statistics. Euclidean distance was chosen as it accommodates any kind of extracted feature vector. We have used statistical measures like X^2 or Kolmogorov Smirnov or frequency-based comparisons such as auto-correlation or Fourier transform comparison [69].

To create a similarity signal, we produce a signal of the rank of the most similar periods to the current period, which in a stable system is expected to be almost constant or linear. This kind of presentation of a signal is useful because if the similarity abruptly changes it indicates a kind of discontinuity in behaviour.

We can combine all these different signal analysis methods to produce different views for the RUPVs described in the next section.

8.5 Recovered Unified Process Views

Recovered Unified Process Views are views of repository data that provide an overview that is similar to the UP diagram in Figure 2.1. These views can be created by plotting signals of development, extracted from the software repositories. Whether the signal consists of counts of commits, mailing list messages, bug reports, or relevant subsets of such artifacts, the default view is to display signals as parallel time-lines of activity.

RUPVs are intended to be analogous to the original UP diagram but instead of proposing a process they display data that is observable and extracted from the repositories used.

8.5.1 Mapping signals back to the Unified Process

Depending on the level of overview that one wants to provide to a stakeholder, a number of related signals will be combined to produce a summary signal. For instance signals about feature requests, signals of feature discussions and signals about story-card creation could be combined into one overview signal about *requirements*. It is these kinds of mapped signals that should be presented to the end user. Before we extract and plot Unified Process signals we have to discuss the matter of observability of UP disciplines based on the data we have.

Observability

We realize that not all signals are observable based on the data we might have available. In terms of our case study we lack concrete requirements, business modelling, design and analysis, and project management signals.

UP Signals

When synthesizing the 9 UP signals we found that much of the data was derived from tool use and the processes being followed. Nonetheless we tried to map what we could back to original UP disciplines. Figure 8.2 demonstrates the UP signals that we extracted from FreeBSD and SQLite. Each signal is described as a sum of events of different signals, summed with a weight of 1 per signal. We have described each UP signal below and use the symbols from Table 8.1 to describe how we produce each signal. We will describe the signals 10 displayed in Figure 8.2:

UP Business Modelling includes requirements, discussion of new features, and client interaction with new features [86]. Based on our data-sets we felt the signals that related to this discipline were commits that mention requirements and commits that mention usability. Unfortunately this misses a lot of important work that might go into a project in order to determine what are its goals. In this sense we feel that this signal does not map well to observable data from FLOSS projects unless one has access to the initial discussions during which the project was fleshed out.

UP Business modelling: $req_{all} + use_{all}$

UP Requirements signals are built up of changes, bugs, and mailing list messages that mention requirements, usability issues, or new functionality. That said all of these signals require lexical approaches as one has to determine if requirements activities are taking place. This produces a signal that is less trustworthy than a signal extracted from a repository of documents that are dedicated to requirements.

symbol	**signal**
req	requirements word-bag
use	usability word-bag
fun	functionality word-bag
anl	analysis word-bag
mtn	maintainability word-bag
eff	efficiency word-bag
rel	reliability word-bag
por	portability word-bag
dep	deployment word-bag
prj	project management word-bag
rel	releases and tags
src	source revisions
bld	build revisions
tst	testing revisions
doc	documentation revisions
rev	all revisions
X_y	X recovered from artifacts y
X_{mrs}	X of commits
X_{bug}	X of bug reports
X_{ml}	X of mailing list messages
X_{all}	X of commits, bug reports and mailing list messages
mtn_{bug}	bug reports that match maintainability word-bag
req_{all}	linear combination of commits, bug reports, and mailing list messages that match the requirements word-bag.

Table 8.1: Symbol table for describing the composition of RUPV signals in Section 8.5.1. Each symbol represents a signal that is made available by previous analysis, these signals can be thought of as time-series. In Section 8.5.1 we combine these signals as linear combinations in order to proxy effort associated with UP related disciplines. The subscripts symbols are meant to specify which artifacts these signals are derived from.

UP Requirements: $req_{all} + use_{all} + fun_{all}$

The UP analysis and design signal might not be easily observable in the projects we were evaluating. Analysis and design will be discussions of new features, drastic changes to architecture and new use cases. During the case study, SQLite did show some design work in terms of designing and implementing the new database file format for SQLite3.

UP Analysis: $anl_{all} + req_{all} + mtn_{all}$

The UP implementation signal is relatively simple; we include revisions that change source code. Of course this could be expanded but we did not want to include all revisions because revisions to build files or documentation are not necessarily implementation.

UP Implementation: src

The UP testing signal is easily extracted from our data sources. Changes to test files, which are easily matched by filename or via lookup table, are a reliable signal to use, but other signals could be added. The UP suggests that testing relates to improvements in qualities such as reliability, functionality, and performance.

UP Testing: $tst + eff_{mrs} + rel_{mrs} + por_{mrs}$

The UP deployment signal is about packaging, portability, distribution and building the software. Our UP deployment signal is a combination of commits that mention deployment, release and tag occurrences, portability related commits, and changes to the build system.

UP Deployment: $dep_{mrs} + por_{all} + bld + rel$

The UP configuration and change management signals consist of commits, and build changes. This signal is a combination of revisions and build system related commits.

UP Configuration: $bld + rev$

The UP project management signal deals with project plans. We did not determine much in our case studies, FreeBSD and SQLite, that concerned project management beyond source control changes that might mention project management. We used a word-bag related to project management in order to find relevant commits, bugs, and discussions.

UP project management: prj_{all}

The UP Environment signal refers to process management and tools. With the data we have from FreeBSD and SQLite the closest signals are the revisions themselves and the build revisions. Commits relating to portability might also be relevant. Some projects have their own tools for cleaning up source code or for generating source code. Perhaps the addition and use of automated tools should be included.

UP Environment: $rev + bld + por_{all}$

We observe that the quality of the RUPVs depends of the available data sources. The raw signals that compose higher level signals, such as the UP signals, are more trustworthy and accurate than these aggregate mappings.

Alternative signals

We found that the signals extracted from the two FLOSS projects we analyzed were not a good conceptual fit for some UP disciplines, so we explored other signals that appeared promising in illustrating the development processes. Alternative process-heavy signals that could also be used are build revision signals and NFR signals.

Build revisions is a signal composed of changes to the build files of a software project. Changes to these build files are often indicative of changes in portability, architecture, and modularity. Portability is often related to build revision changes because supporting new platforms often causes new configuration options or checks to be employed. Architecture is often changed or modified at the same time as build revisions because the build system needs to be informed of new or changed files.

Build revisions: bld

Non functional requirements (NFRs) signals are less accurate than say build revisions. Commits, mailing list messages and bugs that deal with non functional requirements indicate the kind of software quality related topics occurring within a repository. The focus of programmers at certain times on certain qualities are potentially good indicators of process.

NFRs: $use + fun + mtn + eff + rel + por$

We will now discuss how we took all of these signals and produced RUPVs for FreeBSD and SQLite in Sections 8.6 and 8.7.

8.6 FreeBSD Case Study

FreeBSD[4] is a popular open source operating system. FreeBSD was based on the original Berkley Software Distribution (BSD) and BSD386. FreeBSD differs from Linux in that the FreeBSD kernel and userland (UI) are an inseparable package. FreeBSD installations are expected to have at least a bare minimum of programs and tools installed.

Figure 8.2 shows plots (on the left) of 9 disciplines, and build revisions, for FreeBSD over the roughly 16 year history of the project. Figure 8.2 shows a large peak protruding across the FreeBSD UP signals in 2001. In 2001 FreeBSD was ported over to GCC 2.95 as its main compiler in the 4.3 branch of FreeBSD. This meant that much code had to be semi-automatically changed to conform. Mostly the function definitions had to be changed. Requirements-related changes are related to definitions and thus these definition related changes made the peak in 2001 even larger. There were also other requirements-related

[4]FreeBSD: `http://www.freebsd.org/`

changes made as the project was trying to ensure SUSV conformance, especially SUSV2, the Single UNIX Specification version 2.

In Figure 8.2, the bump in 2002 testing, and the minor bumps in deployment, and support environment signals correspond with changes in C coding style with FreeBSD. Much of the FreeBSD project switched from K&R C style declarations to C89 style declarations. This is notable because many of the changes that use the term *conformance* are related to function definition style. This was most likely spurred on by the adoption of GCC3 over GCC2.95. In 2002 there were many portability-related changes as well but there were also two times more releases and CVS tags than in the previous year. When GCC3 was introduced, a new version of binutils was introduced as well, this meant that the project actively had to seek testers to ensure the new GCC and binutils still worked with FreeBSD and the ports software. OpenPAM, a pluggable authentication module was a popular topic and release tag as well.

We investigated the peak in late 2009. At this time, FreeBSD 7.2.0 was released and version 8 of FreeBSD was being prepared. Many requirements-related changes had to do with the merge of Open Source Basic Security Module (OpenBSM). The OpenBSM is designed for security auditing and it was merged into the FreeBSD 8 branch. Since definitions used in the OpenBSM API changed, this merge was flagged as a requirements and analysis related change.

We conclude for these examples that these RUPVs have allowed us to find interesting requirements- and analysis-related behaviours in FreeBSD.

8.7 SQLite Case Study

SQLite is a library-oriented database system. It provides an SQL interface to a common SQLite database file format. It is meant for use in a wide variety of contexts, including embedded devices, such as the iPhone, and web-browsers like Mozilla Firefox. SQLite allows a program to have the power of a SQL driven relational database without the need of a separate DBMS process.

SQLite uses its own source control system called FOSSIL[5]. FOSSIL is a source control system made by the SQLite authors that integrates distributed version control, distributed bug tracking, and a distributed documentation wiki. All of these are combined together into one SQLite file.

SQLite is unusual among FLOSS projects in that the development team explicitly documents, lists, and numbers their requirements[6]. Unfortunately for this study, there

[5]FOSSIL SCM `http://fossil-scm.org/`

[6]SQLite Requirements: `http://www.sqlite.org/requirements.html`

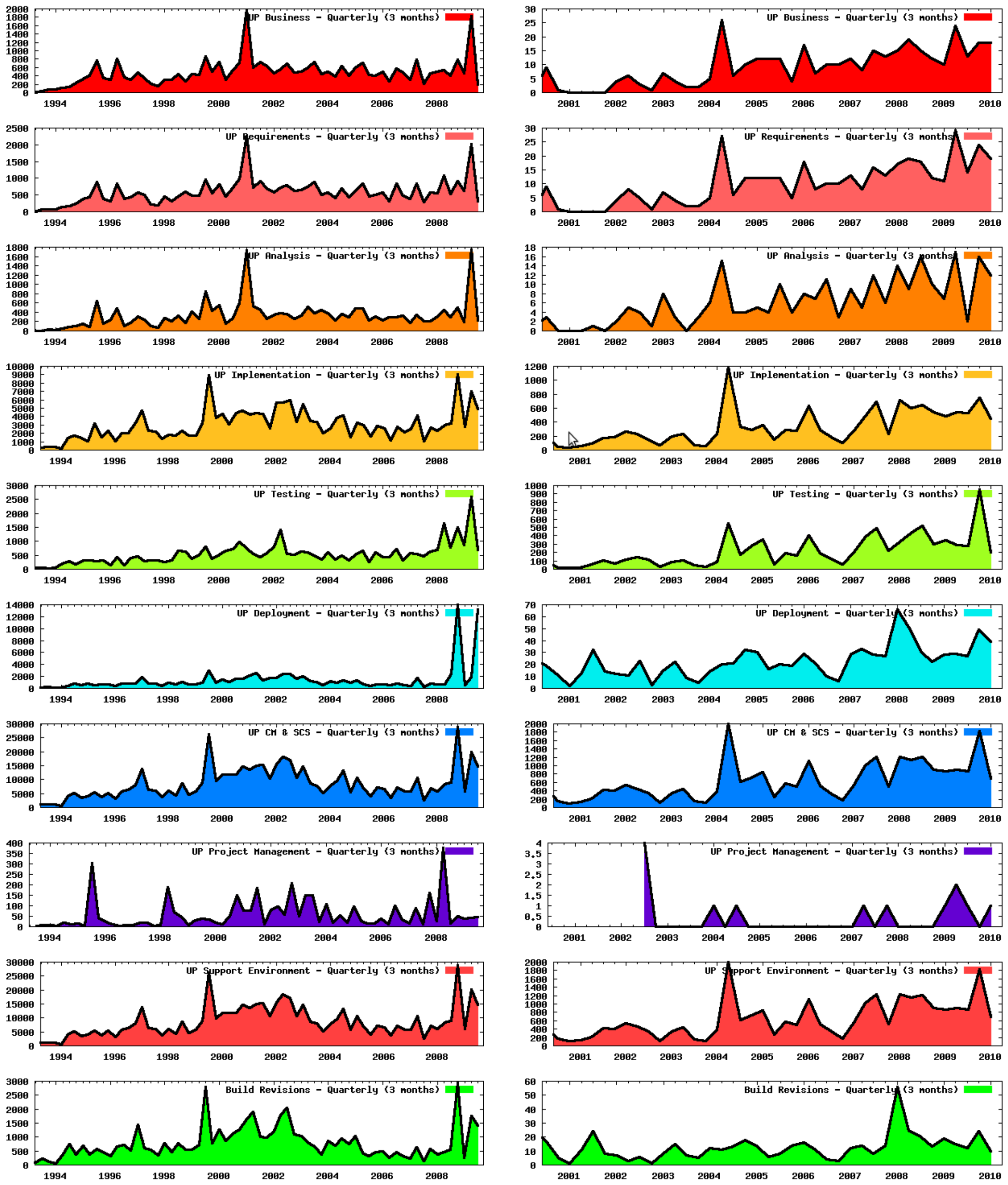


Figure 8.2: Recovered Unified Process Views: These are the Recovered Unified Process Signals of FreeBSD (on the left) and SQLite (on the right): business modelling, requirements, analysis and design, implementation, testing, deployment, configuration management, project management, environment and build revisions.

seems to be little traceability between the commits and the numbered requirements.

To determine if our requirements and analysis signals were at all relevant we decided to investigate SQLite's requirements peak around 2004 (see the right side of Figure 8.2). In our case we did not have any tags in 2004 because when SQLite's CVS repository was converted to a FOSSIL repository, the tags were not imported. The graph, in Figure 8.2 around 2004 has a peak across business analysis, requirements, analysis, implementation, testing, project management, and support environment signals. We can see that a lot of work happened at this time, but what portion was related to requirements? SQLite3 was being developed and there was a need to discuss and implement an appropriate file format. Developers and users were trading implementation notes and referencing literature on database implementations in an attempt to define the SQLite3 file format. One message discussed SQL 92, while a few others discussed literature. Many source control changes were flagged as requirements by the word-bag analysis because they mentioned formats and formatting. The requirements wordlist contains both words and in this case the discussion was actually about the file format itself. Thus this visible peak was attributable to new features and new requirements.

From 2007 to 2008 the requirements, business analysis, and the UP deployment signals peaked. We investigated related events and found that in 2007 that the developers were considering converting their requirements documentation in their code into formal requirements documents. That is the software, SQLite, already worked and met most of the requirements, as well the developers had already tracked the requirements, but this was an effort to improve requirements documentation within the project.

This trend continued into 2009 where requirements were discussed further and API documentation for the C-API of SQLite was being moved out of the code itself and into separate documentation files. This was noticeable in 2009 because requirements-related signals, business analysis, requirements, and analysis, peaked up, but other signals like deployment and testing peaks lagged behind requirements.

Thus for SQLite the behaviour of the requirements-related signals of UP Business Analysis, UP Requirements and UP Analysis actually did indicate that efforts were being made to define a new file format, to formalize requirements, and to move requirements from code into formal documentation.

8.8 Discussion

RUPV's multiple time-line views highlights parts of the development activity allowing us to notice behaviours that might not have been obvious from plain aggregate views such as commits per month. Most notably with SQLite we saw the value of the UP requirements and UP analysis signals in that we were able to find important events that would not have shown up in a commits per month signal.

8.8.1 Threats to validity

There are multiple threats to validity that face this research, we will address them in the context of construct validity, internal validity, external validity and reliability.

With respect to construct validity, we had to address the fact that we rely upon artifacts that programmers create and annotate. In order to improve construct validity we rely on multiple data sources, repositories, and we also use multiple case studies. Multiple data-sources help, but some disciplines we wanted to observe simply are not available in the repositories that we have studied so we had to suggest proxy measures. One construct validity issue we face is that we did not have key informants, the programmers, review our case study results.

Internal validity deals with the fact that we had to infer UP signals. To address this we validated our inferences by building explanations by digging further into the data. Although this does not address the inequality in the size and frequency of different data-sources. As well different projects might not necessarily exhibit the same behaviour and might need largely different mappings. The original Unified Process diagram was about effort per discipline over time we instead showed combinations of events. Two other issues facing our case studies were that we did not address rival explanations very well and did not compare our results against any theoretical model of development.

External validity was addressed by using multiple case studies and showing that we could observe behaviours, such as requirements related work, in both cases. We could have used more case studies and could have been more specific in the questions we asked in each case study.

In terms of reliability the accuracy of the unsupervised and supervised analysis is a concern as it can produce spurious or biased results. The reliability of the supervised analysis could be a concern as its training set required consistent annotation and could suffer from bias.

8.9 Conclusions

We have shown that through the integration of many mining software repositories technologies that one can build high-level views about the software processes of a project. The repositories that we mined were mailing list archives, source control systems and bug tracker systems. Inspired by the UP diagram, shown in Figure 2.1, we wanted to integrate these software repositories to produce an applied version of that diagram: Recovered Unified Process Views (RUPV). RUPVs are one method of software process recovery. We described how to build RUPVs and applied our tools to two open source projects: FreeBSD and SQLite.

In order to attribute behaviour and events to various parallel disciplines we used signal mappings where we combined event and process signals to form proxy signals that represented the events related to a discipline or process. RUPV's multiple time-line views highlight parts of the development activity allowing us to notice behaviours that might not have been obvious from plain aggregate views such as commits per month. These signals proved useful when analyzing FreeBSD and SQLite as they highlighted behaviours, such as requirements and design, that would have been obscured without this kind of analysis. Most notably with SQLite we saw the value of the UP requirements and UP analysis signals in that we were able to find important events that would not have shown up in a commits per month signal.

For managers this work is immediately useful as it could be integrated into project dashboards that show what is going on within a project at a particular time.Stakeholders that benefit from this kind of analysis include: developers who are unfamiliar with the project culture; managers who are somewhat removed from the code; investors interested in the quality of development; and researchers or consultants who want to gain an overview of a project's processes.

8.9.1 Possible Extension

Possible extensions consist of further validating how well the observations presented relate to actual behaviour and processes that take place. We want to interview developers in order to see if RUPVs relate well to their perception of the events that occurred and the processes followed. We also want to further study correlations between signals: we want to see if we can find projects with signals that are not in other projects, such as requirements, and find appropriate proxy signals that might be useful across projects.

Chapter 9

Conclusions

In this thesis, we proposed software process recovery: a combination of mining software repositories, process discovery, process mining, and after-the-fact analysis of development artifacts that recovers a software process. Software process recovery focuses on recovering software development process information from repositories of artifacts that developers use while building software. Developers commit artifacts to these repositories for backup, storage, communication and other purposes. Software process recovery analyzes these artifacts, rather than instrument a live process, in order to recover software development processes. Thus software process recovery can be applied to even dead projects or those without any project members left as it does not rely on interviews or instrumentation. While interviews are valuable they are costly so other alternatives, such as software process recovery, are desirable.

We have confirmed that for the projects that we studied that software process recovery is indeed possible. We have demonstrated that in many cases software development processes can be inferred from the evidence and artifacts that developers leave behind, and thus they are observable and recoverable. Throughout the previous chapters (Chapters 4 to 8) we have repeatedly demonstrated that within the projects we are studied we can detect software development processes. As well, we are capable of attributing the purpose or topic of changes to revisions and commits, bug reports and discussions.

In the previous chapters we illustrated methods of automated and semi-automated software process recovery. We then used many of these techniques to produce Recovered Unified Process Views in Chapter 8.

We analyzed various repositories including: source control systems (also known as version control systems), mailing list archives, and bug trackers. The artifacts within these repositories are often fine-grained and lacking in summaries. One of the primary methods of this work has been to summarize these artifacts whether by summary statistics, case study, or classification. In Chapter 4 we classified revisions by their file types that were associated with software development discipline (source code, testing, documentation, build system). In Chapters 5 and 6 we classified source control commits changes by their maintenance categories. In Chapters 7 and 8 we classified bugs and mailing list discussions that mentioned non-functional requirements.

We found evidence that allowed us to observe that processes occurred:

- In Chapter 4, using source, test, build, and documentation changes we could clearly see that within a project there were a repeating release patterns.
- In Chapter 5 we observe the breakdown of changes by maintenance class. We observed that implementation changes still occur late in a project's life cycle.
- In Chapter 6 we automated the classification of commits into maintenance classes.
- In Chapter 7 we observed that developer topics are often very local and in time but there are many topics which recur across the lifetime of a project. We eventually found that many of the recurrent topics were related to non-functional requirements.
- In Chapter 8 we produced Recovered Unified Process Diagrams that integrated many of the previous chapters and were able to describe how different disciplines were being worked on in different proportions across time.

Who can use or benefit from these techniques? In Section 1.2.1 we discussed a wide variety of stakeholders who could benefit from software process recovery: managers can gain project awareness, especially if they are not intimately involved with programming; those responsible for process certification and validation could use software process recovery to reduce the cost of process documentation and validation; companies could use process recovery to aide ISO 9000 or CMM certification; investors and out-sourcers could use software process recovery to determine how well a team adhered to a particular process; new developers or developers unfamiliar with a code-base could bootstrap their cultural knowledge of a project with process recovery. Thus we have shown there are many motivating applications of software process recovery.

Do our techniques work? Each technique was evaluated and validated against multiple case studies. In most cases these techniques have been published and reviewed via peer review. Essentially each part is validated, leaving only the sum of the parts to be truly validated. There is some attempt to validate the sum. In Chapter 8 we apply the Recovered Unified Process Views to SQLite and FreeBSD in a case study, this is an attempt to validate an application of the integration of techniques. While we feel we have demonstrated the validity of our approaches there are various potential threats to validity.

9.1 Threats to validity

There are three main threats to validity in this research. They are the lack of evidence which hampers the observability of process related events, the reliance on programmers to annotate or describe their actions, and the use of case studies to validate the techniques and methods describe in the previous work. Each of these threats will now be discussed.

9.1.1 Evidence and Observability

Observability of process is an important issue facing this research, as certain aspects of a project are often implicit or unobservable as they lack necessary evidence. For instance face to face meetings are often not recorded and cannot be extracted. As well some projects have implicit requirements because they are meant to mimic the functionality of existing products. We found that business analysis and project management events were particularly hard to recover from the artifacts we analyzed [81]. The quality of the data in the project determines the observability of the software development processes that were used.

We found many parts of the process were only partially observable that could threaten the validity of this research:

- Business Modelling — many projects have clear business goals before they start and these are not clearly recorded. It is also difficult to tease our business related goals from discussions.
- Requirements — most projects do not have clear requirements documentation. Many FLOSS projects are based on existing systems and thus the requirements are already clear.
- Design — distinguishing design from requirements is often difficult. Design is not always apparent. Sometimes it can be captured in discussions but in FLOSS project design documents often do not exist.
- Reasons for changes — the rationale or motivation behind a change is not always documented and thus has to be inferred.

Many of these observability issues are related to the level of detail within the artifacts.

9.1.2 Detail of Fine Grained Artifacts

Another threat to the validity of the research is the quality and level of detail that exists within the artifacts that we study and rely upon. We rely on the artifacts that developers create. Developers do not create these artifacts in order to enable research. These artifacts are meant to help the development of a software project. Thus the annotations used by programmers might not be enough to determine the actual behaviour. Developer behaviour can change over time. Programmers can phase annotate and describe their changes inconsistently. Thus much analysis depends on the context, the programmers, and the project's underlying culture.

9.1.3 Validating with Case Studies

Our validations of the techniques that we discussed in previous chapters mostly consisted of case studies. Often we would create training sets based on real data in repositories, the classification of this data was done manually by hand and using our own judgement. This implies that some of these results might not be as generalizable as we hope, thus we have to approach each project distinctly and evaluate the context of the project and the availability of its artifacts.

9.1.4 Cost of Machine Learning

Some of this work relies on machine learners that require training sets. Supervised machine learning is actually quite costly to deploy as it requires a training set. In some cases it might require the expertise of the developer in order to help build a training set. If it takes 2 minutes to annotate 1 entry, 120 entries takes 4 hours of annotation time. Thus building training sets is costly and could harm our claim of interview-free analysis.

9.1.5 Summary

There are various possible threats to validity but we have attempted to validate each part of this thesis with case studies on real systems so we feel confident that while threats do exist and some studies could be improved, what we have found is a good indication that software process recovery can recover useful information about the de facto software development processes of a project.

9.2 Future Work

This work is an initial attempt at combining much of our research and MSR research in order to recover software processes. Thus we could not cover all angles. We mostly focused on analyzing revisions and gave some attention to the analysis of bug reports and mailing list archive discussions.

The future work in software process recovery is broad and includes:

- How to analyze documentation artifacts with respect to software process recovery? The documents and document repositories of particular interest are requirements and design documents. These repositories and documents were less explicit during our studies because they are relatively uncommon in FLOSS projects.
- Case studies of commercial systems, in particular what are the repositories that are available to commercial systems that we could leverage. These repositories might include various sorts of logs and field reports.
- Investigating if various agile method techniques such as pair programming are observable with our methods.
- Process validation and verification, we focused mostly on finding the underlying process but we did not attempt to validate against prescribed or existing processes.
- How to identify phases? We suspect that phases are combinations of behaviours that can be learned and extracted, thus one could identify a phase by how close it was to other identified phases.

This is a subset of possible future directions. Software process recovery is not a problem that has been solved in this thesis. A small part of it has been explored and we suggest there is a broad field of research ahead of us.

9.3 Summary

Software process recovery is the recovering of software development processes from projects by analyzing the development artifacts, such as source code revisions and discussions about patches, that developers and users create and leave behind. Software process recovery is an after-the-fact analysis of software artifacts that infers underlying software development processes.

In this thesis we proposed to investigate multiple aspects of software process recovery. We showed that in many cases software development processes were observable, we showed we could recover some of them. We demonstrated many methods of attributing purpose, behaviour and topic labels to many kinds of artifacts. These labels, attributions, classifications and tags allow us to characterize the purposes and behaviour behind the events we are analyzing. This allows us to determine what the observable software development process was. We validated our approaches to software process recovery with many case studies on multiple FLOSS project.

We motivated the need for software process recovery by showing how a variety of stakeholders, such as managers, developers, and potential investors, could benefit from software recovery for a variety of purposes, such as ISO 9000 certification and project awareness.

We proposed software process recovery. We proposed a number of methods and techniques to recover specific aspects of software development processes. We then integrated these techniques to produce a general overview: Recovered Unified Process Views. We are confident that the combination of these proposed techniques and their integration confirms that we can recover some software development processes from these artifacts. This thesis is a preliminary work in the field of software process recovery; there is much to do and still much MSR-related research to apply to software process recovery.

References

[1] Capability maturity model for software. Technical Report CMU/SEI-91-TR-24 ADA240603, 1991.

[2] *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*. IEEE Computer Society, 2006.

[3] *NIST/SEMATECH e-Handbook of Statistical Methods*, 2008. http://www.itl.nist.gov/div898/handbook/.

[4] Tanaka Akira. Cvssuck - inefficient CVS repository grabber. http://cvs.m17n.org/ akr/cvssuck/.

[5] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. What's a typical commit? a characterization of open source software repositories. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 182–191, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Juan Jose Amor, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Discriminating development activities in versioning systems: A case study. In *Proceedings PROMISE 2006: 2nd. International Workshop on Predictor Models in Software Engineering*, 2006.

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

[8] Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya. A theory of aspects as latent topics. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 543–562, Nashville, 2008.

[9] Thomas Ball, Jung-Min Kim Adam, A. Porter Harvey, and P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

[10] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 1st edition, October 1999.

[11] David Bellin, Manish Tyagi, and Maurice Tyler. Object-oriented metrics: an overview. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 4. IBM Press, 1994.

[12] Dirk Beyer and Ahmed E. Hassan. Animated Visualization of Software History using Evolution Storyboards. In *WCRE* [2], pages 199–210.

[13] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining Email Social Networks. In *Proceedings of the Third International Workshop on Mining software repositories*, pages 137–143. ACM, 2006.

[14] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open Borders? Immigration in Open Source Projects. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 6. IEEE Computer Society, 2007.

[15] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Putting it All Together: Using Socio-Technical Networks to Predict Failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2009.

[16] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent Social Structure in Open Source Projects. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–35. ACM, 2008.

[17] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[18] David M. Blei, Andrew Y. Ng, and Michael I Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3(4-5):993–1022, May 2003.

[19] Jørgen Bøegh. A New Standard for Quality Requirements. *IEEE Software*, 25(2):57–63, 2008.

[20] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, 1986.

[21] Barry Boehm, J R Brown, and M Lipow. Quantitative Evaluation of Software Quality. In *International Conference on Software Engineering*, pages 592–605, 1976.

[22] J. Brichau, C. De Roover, and K. Mens. Open unification for program query languages. In *Chilean Society of Computer Science, 2007. SCCC '07. XXVI International Conference of the*, pages 92–101, Nov. 2007.

[23] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. Characteristics of open source projects. volume 00, page 317, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[24] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. Characteristics of open source projects. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 317, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Lawrence Chung, Brian A Nixon, Eric S Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*, volume 5 of *International Series in Software Engineering*. Kluwer Academic Publishers, Boston, October 1999.

[26] Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. The Detection and Classification of Non-Functional Requirements with Application to Early Aspects. In *International Requirements Engineering Conference*, pages 39–48, Minneapolis, Minnesota, 2006.

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

[28] Jonathan E. Cook. *Process Discovery and Validation through Event-Data Analysis*. PhD thesis, Computer Science Dept., University of Colorado, 1996.

[29] Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 73–82, New York, NY, USA, 1995. ACM Press.

[30] Jonathan E. Cook and Alexander L. Wolf. Balboa: A framework for event-based process data analysis. In *Proc. of the 5th International Conference on the Software Process*, pages 99–110, June 1998.

[31] Jonathan E. Cook and Alexander L. Wolf. Software process validation: quantitatively measuring the correspondence of a process to a model. *ACM Trans. Softw. Eng. Methodol.*, 8(2):147–176, 1999.

[32] Davor Cubranic and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 2003 International Conference on Software Engineering*, pages 408–418, Portland, 2003. Association for Computing Machinery.

[33] Marco D'Ambros and Michele Lanza. Reverse Engineering with Logical Coupling. In *WCRE* [2], pages 189–198.

[34] Kresimir Delac, Mislav Grgic, and Sonja Grgic. Independent comparative study of PCA, ICA, and LDA on the FERET data set. *International Journal of Imaging Systems and Technology*, 15(5):252–260, 2006.

[35] Dirk Draheim and Lukasz Pekacki. Process-centric analytical processing of version control data. In *Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 131–136. IEEE, 2003.

[36] Stéphane Ducasse, Michele Lanza, Andrian Marcus, Jonathan I. Maletic, and Margaret-Anne D. Storey, editors. *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, September 25, 2005, Budapest, Hungary.* IEEE Computer Society, 2005.

[37] Neil A Ernst and John Mylopoulos. On the perception of software quality requirements during the project lifecycle. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, Essen, Germany, June 2010. in press.

[38] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database.* MIT Press, 1998.

[39] Norman E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman & Hall, Ltd., 1991.

[40] Michael Fischer and Harald Gall. EvoGraph: A Lightweight Approach to Evolutionary and Structural Analysis of Large Software Systems. In *WCRE* [2], pages 179–188.

[41] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, 2003.

[42] Harald Gall, Mehdi Jazayeri, , and Jacek Krajewski. CVS Release History Data for Detecting Logical Couplings. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, September 1998.

[43] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS Release History Data for Detecting Logical Couplings. In *Proc. of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 12–23, 2003.

[44] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 99, Washington, DC, USA, 1999. IEEE Computer Society.

[45] D. M. German. Decentralized open source global software development, the GNOME experience. *Journal of Software Process: Improvement and Practice*, 8(4):201–215.

[46] D. M. German. Mining CVS repositories, the softChange experience. In *1st International Workshop on Mining Software Repositories*, pages 17–21, May 2004.

[47] D. M. German. A study of the contributors of PostgreSQL. In *3rd International Workshop on Mining Software Repositories–MSR Challenge Reports (MSR 2006)*, May 2006. Received *Best Challenge Report* Award.

[48] Daniel M. German and Abram Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Proceedings of 11th International Software Metrics Symposium (Metrics 2005)*, 2005.

[49] Daniel M. German, Abram Hindle, and Norman Jordan. Visualizing the evolution of software using softchange. In *Proceedings SEKE 2004 The 16th Internation Conference on Software Engineering and Knowledge Engineering*, pages 336–341, 3420 Main St. Skokie IL 60076, USA, June 2004. Knowledge Systems Institute.

[50] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *ICSM*, pages 40–49, 2004.

[51] Michael W. Godfrey and Qiang Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of International Conference on Software Maintenance*, pages 131–142, 2000.

[52] Scott Grant, James R. Cordy, and David Skillicorn. Automated concept location using independent component analysis. In *15th Working Conference on Reverse Engineering*, 2008.

[53] Jin Guo, King Chun Foo, Liliane Barbour, and Ying Zou. A business process explorer: recovering and visualizing e-commerce business processes. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 871–874, New York, NY, USA, 2008. ACM.

[54] Jin Guo and Ying Zou. A business process explorer: Recovering business processes from business applications. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 333–334, Oct. 2008.

[55] Jin Guo and Ying Zou. Detecting clones in business applications. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 91–100, Oct. 2008.

[56] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[57] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[58] R. Harrison, S.J. Counsell, and R.V. Nithi. An evaluation of the mood set of object-oriented software metrics. *Software Engineering, IEEE Transactions on*, 24(6):491–496, Jun 1998.

[59] Deborah Hartmann and Robin Dymond. Appropriate agile measurement: Using metrics and diagnostics to deliver business value. In *AGILE '06: Proceedings of the conference on AGILE 2006*, pages 126–134, Washington, DC, USA, 2006. IEEE Computer Society.

[60] Ahmed Hassan. Mining software repositories to guide software development. `http://plg.uwaterloo.ca/~aeehassa/home/pubs.html`, 2005. Accessed July.

[61] Ahmed E. Hassan and Richard C. Holt. C-REX: An Evolutionary Code Extractor for C - (PDF). Technical report, University of Waterloo. http://plg.uwaterloo.ca/ aeehassa/home/pubs/crex.pdf.

[62] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *Proceedings of ICSM 2004: International Conference on Software Maintenance*, pages 284–293, September 2004.

[63] W. Heijstek and M.R.V. Chaudron. Evaluating rup software development processes through visualization of effort distribution. In *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*, pages 266 –273, 3-5 2008.

[64] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Forecasting the number of changes in eclipse using time series analysis. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 32, Washington, DC, USA, 2007. IEEE Computer Society.

[65] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 21, Washington, DC, USA, 2007. IEEE Computer Society.

[66] Jim Highsmith and Martin Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[67] Emily Hill. Developing natural language-based program analyses and tools to expedite software maintenance. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 1015–1018, New York, NY, USA, 2008. ACM.

[68] A. Hindle, M.W. Godfrey, and R.C. Holt. Release pattern discovery: A case study of database systems. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 285–294, Oct. 2007.

[69] A. Hindle, M.W. Godfrey, and R.C. Holt. Mining recurrent activities: Fourier analysis of change events. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 295–298, May 2009.

[70] A. Hindle, Zhen Ming Jiang, W. Koleilat, M.W. Godfrey, and R.C. Holt. Yarn: Animating software evolution. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 129–136, June 2007.

[71] Abram Hindle, Neil Ernst, Michael W. Godfrey, Richard C. Holt, and John Mylopoulos. What's in a name? on the automated topic naming of software maintenance activities. In submission: `http://softwareprocess.es/whats-in-a-name`, 2010.

[72] Abram Hindle and Daniel M. German. Scql: a formal model and a query language for source control repositories. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[73] Abram Hindle, Daniel M. German, Michael W. Godfrey, and Richard C. Holt. Automatic classification of large changes into maintenance categories. In *International Conference on Program Comprehension*, Vancouver, 2009.

[74] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

[75] Abram Hindle, Michael Godfrey, and Ric Holt. Release Pattern Discovery via Partitioning: Methodology and Case Study. In *Proceedings of the Mining Software Repositories 2007.* IEEE Computer Society, 2007.

[76] Abram Hindle, Michael Godfrey, and Ric Holt. From indentation shapes to code structures. In *8th IEEE Intl. Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*, 09 2008.

[77] Abram Hindle, Michael Godfrey, and Ric Holt. Reading beside the lines: Indentation as a proxy for complexity metrics. In *Proceedings of ICPC 2008*, June 2008.

[78] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Release Pattern Discovery via Partitioning: Methodology and Case Study. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 19, Washington, DC, USA, 2007. IEEE Computer Society.

[79] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming*, 74(7):414 – 429, 2009. Special Issue on Program Comprehension (ICPC 2008).

[80] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. What's hot and what's not: Windowed developer topic analysis. In *International Conference on Software Maintenance*, pages 339–348, Edmonton, Alberta, Canada, September 2009.

[81] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Software process recovery using recovered unified process views. In *Proceedings of the International Conference on Software Maintenance 2010 (ICSM 2010)*, 2010.

[82] Abram Hindle, ZhenMing Jiang, Walid Koleilat, Michael W. Godfrey, and Richard C. Holt. Yarn ball example, 2007. http://swag.uwaterloo.ca/˜ahindle/yarn/postgres.html.

[83] Thomas Hofmann. Probabilistic latent semantic indexing. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 50–57, New York, NY, USA, 1999. ACM.

[84] Eibe Frank Ian H.Witten. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor, October 1999.

[85] Software engineering – Product quality – Part 1: Quality model. Technical report, International Standards Organization - JTC 1/SC 7, 2001.

[86] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[87] Chris Jensen and Walt Scacchi. Simulating an automated approach to discovery and modeling of open source software development processes. In *Proceedings of the Third Workshop on Open Source Software Engineering ICSE03-OSSE03*, May 2003.

[88] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, 2007.

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

[90] Paul Kantor. Foundations of statistical natural language processing. *Inf. Retr.*, 4(1):80–81, 2001.

[91] A. Kayed, N. Hirzalla, A.A. Samhan, and M. Alfayoumi. Towards an ontology for software product quality attributes. In *International Conference on Internet and Web Applications and Services*, pages 200–204, May 2009.

[92] Chris F. Kemerer and Sandra Slaughter. An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Softwware Engineering*, 25(4):493–509, 1999.

[93] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[94] Sunghun Kim, Jr. E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March/April 2008.

[95] A.J. Ko, B.A. Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 127–134, Sept. 2006.

[96] A. Kuhn, S. Ducasse, and T. Girba. Enriching reverse engineering with semantic clustering. *Reverse Engineering, 12th Working Conference on*, pages 10 pp.–, Nov. 2005.

[97] Hsiang-Jui Kung. Quantitative method to determine software maintenance life cycle. In *ICSM*, pages 232–241. IEEE Computer Society, 2004.

[98] Meir M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[99] Meir M. Lehman. Laws of software evolution revisited. 1998.

[100] Meir M Lehman, J F Ramil, P D Wernick, D E Perry, and W M Turski. Metrics and laws of software evolution-the nineties view. In *International Software Metrics Symposium*, pages 20–32, Albuquerque, NM, 1997.

[101] Semmle Limited. Semmle company website. `http://semmle.com`.

[102] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining concepts from code with probabilistic topic models. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 461–464, New York, NY, USA, 2007. ACM.

[103] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining Eclipse Developer Contributions via Author-Topic Models. *International Workshop on Mining Software Repositories at ICSE*, pages 30–30, May 2007.

[104] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining internet-scale software repositories. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA, 2008.

[105] Y. Liu and E. Stroulia. Reverse Engineering the Process of Small Novice Software Teams. In *Proc. 10th Working Conference on Reverse Engineering*, pages 102–112. IEEE Press, November 2003.

[106] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Working Conference on Reverse Engineering*, pages 155–164, Antwerp, Belgium, 2008.

[107] Mircea Lungu and Michele Lanza. Exploring Inter-Module Relationships in Evolving Software Systems. *CSMR 2007*, 0:91–102, 2007.

[108] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 33–42, May 2005.

[109] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223, Nov. 2004.

[110] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D Representations for Software Visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, New York, NY, USA, 2003. ACM Press.

[111] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12):1415–1425, 1989.

[112] J McCall. *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager*, volume 1-3. General Electric, November 1977.

[113] Qiaozhu Mei, Xuehua Shen, and ChengXiang Zhai. Automatic labeling of multinomial topic models. In *International conference on Knowledge discovery and data mining*, pages 490–499, San Jose, California, 2007.

[114] Roberto Meli. Measuring change requests to support effective project management practices. In *ESCOM Conference*, 2001.

[115] Tom Mens and Serge Demeyer. Evolution metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001. ACM Press.

[116] Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 83–86, New York, NY, USA, 2001. ACM Press.

[117] Tom Mens, Juan Fernandez-Ramil, and S Degrandsart. The evolution of Eclipse. In *International Conference on Software Maintenance*, pages 386–395, Shanghai, China, October 2008.

[118] Cédric Mesnage and Michele Lanza. White Coats: Web-Visualization of Evolving Software in 3D. In Ducasse et al. [36], pages 40–45.

[119] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 11–20, May 2009.

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

[121] Audris Mockus, Roy T. Fielding, and James Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[122] Hausi A. Muller and K. Klashinsky. Rigi - A System for Programming-in-the-large. Technical report, In IEEE 10th International Conference on Software Engineering(ICSE-1998), April 1998.

[123] I. Myers and M. H. McCauley. *A Guide to the Development and use of the Myers-Briggs Type Indicator*. Consulting Psychologists Press, 1985.

[124] Michael Ogawa, Kwan-Liu Ma, Christian Bird, Premkumar T. Devanbu, and Alex Gourley. Visualizing Social Interaction in Open Source Software Projects. In *Sixth International Asia-Pacific Symposium on Visualization*, pages 25–32, 2007.

[125] Paul W. Oman and Jack Hagemeister. Construction and testing of polynomials predicting software maintainability. *J. Syst. Softw.*, 24(3):251–266, 1994.

[126] M. C. Paulk, B. Curtis, E. Averill, J. Bamberger, T. Kasse, M. Konrad, J. Perdue, C. Weber, and J. Withey. *The capability maturity model: guidelines for improving the software process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[127] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing Multiple Evolution Metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 67–75, St. Louis, Missouri, 2005. ACM Press.

[128] Denys Poshyvanyk and Andrian Marcus. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *International Conference on Program Comprehension*, pages 37–48, June 2007.

[129] Denys Poshyvanyk, Andrian Marcus, Vaclav Rajlich, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. *International Conference on Program Comprehension*, 0:137–148, 2006.

[130] Sandeep Purao and Vijay Vaishnavi. Product metrics for object oriented systems. *ACM Computing Surveys*, 35(2), 2003.

[131] Ranjith Purushothaman. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005. Member-Dewayne E. Perry.

[132] Jacek Ratzinger, Michael Fischer, and Harald Gall. EvoLens: Lens-View Visualizations of Evolution Data. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–112, Lisbon, Portugal, September 2005. IEEE Computer Society Press.

[133] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary (O'Reilly Linux)*. O'Reilly, October 1999.

[134] Peter C. Rigby and Ahmed E. Hassan. What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list. *Mining Software Repositories, International Workshop on*, 0:23, 2007.

[135] G. Ripoche and L. Gasser. Scalable automatic extraction of process models for understanding F/OSS bug repair. In *Proceedings of the International Conference on Software and Systems Engineering and their Applications (ICSSEA'03)*, December 2003.

[136] Gregorio Robles, Jesus M. Gonzalez-Barahona, Daniel Izquierdo-Cortazar, and Israel Herraiz. Tools for the study of the usual data sources found in libre software projects. *International Journal of Open Source Software and Processes*, 1(1):24–45, Jan-March 2009.

[137] J. Rosenberg. Some misconceptions about lines of code. *metrics*, 00:137, 1997.

[138] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–339. IEEE Computer Society Press, 1987.

[139] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[140] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[141] Judith D. Singer and John B. Willett. *Applied Longitudinal Data Analysis: Modeling Change and Event Occurence*. Oxford University Press, 2003.

[142] H. Siy, P. Chundi, D.J. Rosenkrantz, and M. Subramaniam. Discovering dynamic developer relationships from software version histories by time series segmentation. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 415–424, Oct. 2007.

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

[144] Iso Iec Software and Raghu Singh. International standard. In *Software Lifecycle Process Standards, in CrossTalk*, pages 6–8, 1989.

[145] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 123–132, June 2008.

[146] Letha H. Etzkorn Stacy K. Lukins, Nicholas A. Kraft. Source code retrieval for bug localization using latent dirichlet allocation. In *15th Working Conference on Reverse Engineering*, 2008.

[147] Dirk Stelzer, Werner Mellis, and Georg Herzwurm. A critical look at iso 9000 for software quality management. *Software Quality Control*, 6(2):65–79, 1997.

[148] Margaret-Anne D. Storey, Casey Best, and Jeff Michaud. SHriMP Views: An Interactive Environment for Exploring Java Programs. In *9th International Workshop on Program Comprehension (IWPC 2001), 12-13 May 2001, Toronto, Canada*, pages 111–112. IEEE Computer Society, 2001.

[149] E. Burton Swanson. The Dimensions of Maintenance. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[150] Nikita Synytskyy, Richard C. Holt, and Ian Davis. Browsing software architectures with lsedit. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 176–178, Washington, DC, USA, 2005. IEEE Computer Society.

[151] Alexandru Telea and Lucian Voinea. Interactive Visual Mechanisms for Exploring Source Code Evolution. In Ducasse et al. [36], pages 52–57.

[152] Christoph Treude and Margaret-Anne Storey. ConcernLines: A timeline view of co-occurring concerns. In *International Conference on Software Engineering*, pages 575–578, Vancouver, May 2009.

[153] Andrew Tridgell, Paul Mackerras, and Wayne Davison. Rsync. http://www.samba.org/rsync/.

[154] G. Tsoumakas, I. Katakis, and I. Vlahavas. Mining multi-label data. In O. Maimon and L. Rokach, editors, *Data Mining and Knowledge Discovery Handbook*. Spring, 2nd edition, 2010.

[155] Wladyslaw M. Turski. Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering*, 22(8):599–600, 1996.

[156] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47(2):237 – 267, 2003.

[157] Wil M. P. van der Aalst and Kristian Bisgaard Lassen. Translating unstructured workflow processes to readable bpel: Theory and implementation. *Inf. Softw. Technol.*, 50(3):131–159, 2008.

[158] Stanley Wasserman, Katherine Faust, and Dawn Iacobucci. *Social Network Analysis: Methods and Applications (Structural Analysis in the Social Sciences).* Cambridge University Press, November 1994.

[159] Timo Wolf, Adrian Schr, Daniela Damian, Lucas D. Panjer, and Thanh H.D. Nguyen. Mining task-based social networks to explore collaboration in software teams. *IEEE Software*, 26(1):58–66, 2009.

[160] Jingwei Wu and Richard C. Holt. Linker-Based Program Extraction and Its Uses in Studying Software Evolution. Technical report, In Proceedings of the International Workshop on Unanticipated Software Evolution (FUSE 2004), March 2004.

[161] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 220–229, Washington, DC, USA, 2008. IEEE Computer Society.

[162] Thomas Zimmermann and Peter Weisgerber. Preprocessing CVS data for fine-grained analysis. In *1st International Workshop on Mining Software Repositories*, May 2004.