Documentation Practices in Open Source – A Study of Apache Derby

Master's Thesis

Håvard Mork

NTNU
Norwegian University of
Science and Technology

Abstract

Open source is one of the more interesting trends in software engineering today. The goal of the software engineering discipline is to increase efficiency in the development process, and maximize quality of the product. Open source development processes offer the potential for reducing costs for commercial enterprises.

This master's thesis addresses how open source documents architecture, and how it uses documentation in general. Open source has a reputation of creating high quality software, but documentation of process and product is weak. This may be a hurdle for wider adoption of open source processes, as a thorough understanding of a product's qualities is central to its success. The goal is to better understand documentation requirements in open source. The study is based on participation to the Apache Derby open source project. Action research is the research method.

The findings show that the Apache Derby documents its artifacts in a number of ways, but fails to aggregate it in a meaningful way. A rich set of written communication mediums compensate for this by giving developers the ability to understand the product over time. The study suggests the popularity and diffusion of an open source project may affect requirements for documentation.

Preface

This master's thesis is the result of one semester's work, from January to June 2006, and concludes the requirements for a master's degree in computer science at the Norwegian University of Science and Technology (NTNU), Department of Computer and Information Science (IDI).

In this thesis, I have used participation to an open source project to better understand open source development processes, getting experience in software engineering practices, while also trying to enhance programming skills through observing and working with the community. The work in this thesis follows in the footsteps of a directed study from autumn 2005 which aimed to investigate leadership issues in commercially controlled open source projects [Mor05].

The precise focus of the thesis is investigating which documentation practices exist in one open source project, and through participation to this project also reflecting on the challenges for documentation or architectural descriptions in open source. Software engineering literature presses the need for documenting software, not only for its users, but notably also for the developers that will maintain the software, i.e. [CBB+02].

This master's thesis is based on the following assignment text, modified to encompass my understanding of the task:

> *Candidates will have to participate to one open source software (OSS) project in order to learn about open source in a software engineering context, improve programming skills, and contribute to the discussion around OSS. The candidate can influence the research focus, as well as the project to work with. The research questions will have to be grounded in OSS literature.*

I would like to thank my supervisor, Professor Letizia Jaccheri for her guidance through my work on the master's thesis. Also thanks to the many people at the Department of Computer and Information Science who have contributed through valuable discussion, and helped make this a memorable semester.

Trondheim, 22nd June 2006

Håvard Mork
havard.mork@gmail.com

Contents

Chapter 1

Introduction

Software engineering research addresses the need to find ways to make software development more efficient and create higher quality software systems. The set of disciplined methods used in software engineering development processes aim to allow a software system to grow and develop into a final form that is useful for the stakeholders of the system. The *open source* software engineering process is the focus in this master's thesis.

Documentation is commonly viewed as artifacts only necessary for users of software. Looking beyond this, it can be observed that documentation is a separate project artifacts that needs development and maintenance in order to be useful and survive. Documentation has both prescriptive and descriptive purposes in a software system [CBB+02], and is in most development processes regarded as essential. However, while the production of documentation is seen as axiomatic, little is known about what type of documentation is regarded as useful for developers [STT01].

The research goal of the master's thesis is to highlight relations between how software documentation and architectural descriptions relate to the success of the open source development method. Certain aspects of this will be highlighted through participation to an open source project, the Apache Derby database system.

This chapter will further define the goals of this project, its context, and how this report is structured.

1.1 Open source

Open source has become a cultural phenomenon which also is becoming common in corporate environments. It stems from a hacker culture in the 1960s and 70s, where it was created out of the need for computer enthusiasts (*hackers*) to share code among themselves. In the mid-80s, the Free Software Foundation was started in order to promote distribution and production of free software. The early efforts were primarily founded in idealistic motives, that software should be free for users. This was a response to the increasing commercialization of computer software and operating systems.

In many ways, open source is light-weight in comparison to commercially used soft-

ware development processes. The primary characteristic of open source development processes is the openness. Many features of commercial software development are lacking in open source, such as requirements engineering with the intended customers, formal use of architecture descriptions, and formal release planning. Open source travels light by not requiring some of the control mechanisms typically associated with software engineering, but still accomplishes acceptable results through a 'scratch-own-itch', peer-recognizing culture of sharing.

Successes of the open source development method, according to the number of users, include operating systems such as Linux and FreeBSD. Popular desktop applications have also gained widespread diffusion such as Eclipse, OpenOffice, and the Firefox web browser. Open source is traditionally used to develop software *by developers, for developers*. The trend is now moving towards development of end-user software in a market which is already saturated with commercial alternatives. This trend of commoditization is in some ways an effect of open source, but it is also the goal of the original free software movement, which is giving freedom to the users.

Open source development usually is centered around a community of people working on a common goal. The community may be composed both of volunteers that work on open source out of a personal need or interest, and professionals that are paid to work on it. The economic foundation can therefore not be explained with only a private investment model, but must include the deeper cultural and social roots [HNH03].

In order to use open source in a commercial context, it is not only necessary to understand how code is produced, but in order to be successful it is also necessary to make the adaptions to the cultural and social aspects of open source.

1.2 The software engineering context

Software engineering is according to IEEE Standard 610.12 defined as *"the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software"*, and also the study of approaches for applying this [IEE90]. It is an engineering discipline incepted in the 1960s from the need to address problems like cost overruns, quality and schedule problems [WRH+00].

Various approaches exist for applying software engineering to real-world projects in order to accomplish the goals of quality and cost saving. These are referred to as development processes. Some software development processes that are commonly used are the Waterfall model, Extreme Programming, and Agile programming [Koc04]. These models have in common that they support the software development through introducing well-defined phases, force programmers to drop their tendency to go on coding with no plan, and instead follow through with a rigorous set of activities like planning, testing, and deployment.

Open source is strictly speaking not *in itself* a software development process like the waterfall model. Strictly speaking, most implementations of open source will even struggle to fit into IEEE's definition of software engineering. It does not prescribe the phases

the software should go through, or its planning or testing practices. Open source is a set of principles for a development process [Koc04]. These are that the software's license should abide by the Open Source Definition [Ini05], which i.e. requires the software's source code should be freely available together with any copies of the program.

The freedom of open source the most distinctive factor when comparing commercial and open source software. This does not imply they are opposites in a software engineering context, but merely two approaches for reaching software objectives. Open source has been described as a cycle of innovation in which those that have the skills share ideas and build on each other's work [Woo05]. The Open Source Initiative, however, claims that *"[Open source] produces better software than the traditional closed model, in which only a very few programmers can see the source and everybody else must blindly use an opaque block of bits."* [1]. Whether this is true or not, a discussion of which is the better development process is meaningless in an industry where there is a need for both.

Other aspects that are important for software engineering include themes such as architecture, development environments, programming languages, databases technology, software evolution, and configuration management. For this thesis, the focus will be documentation artifacts in software development. The production and maintenance of documentation as separate deliverables in a project is important for adoption of software [BCK03, MD04]. Documentation for developers is the primary concern of this thesis.

1.3 Problem description

This master's thesis aims to determine how open source software projects produce and maintain documentation. Freely available open source programs are increasingly being used as off-the-shelf components in commercial software development in order to reduce cost and time-to-market. The requirements for such off-the-shelf reuse include documenting the software thoroughly, following open standards, and easing the evaluation and adoption of the open source software component.

The study is based on participation to the Java database system Apache Derby[2]. The participation will aid in understanding these practices in open source. A better understanding of how Apache Derby implements the open source development method is expected from the study, as well as experience in contributing to an open source project. Apache Derby has been considered interesting for the purposes of this thesis due to its code complexity, and its relatively widespread usage.

The typical goal for open source software is to create a system which is useful or interesting to those who are working on it, while still allowing its users to adapt or evolve the system into their needs [GT00]. For embedded components such as Apache Derby, its success is dependent on having an API[3] that simplifies integration with other

[1] http://www.opensource.org/

[2] http://db.apache.org/derby/

[3] Application Programming Interface, a structure or interface that allows communication between parts of a system.

systems. Having standardized interfaces with proper specifications is necessary in order for the component to be usable for adopters [CBB+02]. New contributors to open source will also benefit from the existence of specifications.

Open source literature has previously pressed the need to highlight what characteristics is conductive to smooth open source development [LT00]. Models and frameworks in literature highlight the process aspects of open source development [DSV03], as well as structural and cultural [SSR02]. This thesis is an effort to understand the gap between open source and software architecture by focusing on how documentation practices in open source is affected by adopters' and developers' needs.

Participation and contribution to the Apache Derby project is expected to aid in the understanding of existing practices in open source on these matters. The practical take on this issue can also give insight into other problems facing the open source development method with regard to documentation. Research questions posed in this study are:

How is documentation maintained in the Apache Derby project?
The existence of documentation in open source projects may be a requirement for both adopters and developers of open source components. Developers choosing to extend or adopt open source components may require sources of documentation that explains the architecture and design choices.

Using a documented software architecture throughout the lifetime of a software system can improve the quality and maintainability of the system [BHB99], and thereby reduce cost and avoid inconsistencies in the software. Importance of documentation in open source development is also briefly addressed by [BR02, VV04].

Studying how open source deals with documentation further may be helpful to predict success criteria for the use of an open source component. Open source may by experts be regarded as black box systems that needs no further explanation, but documentation, especially architectural documentation holds the key to post-deployment system understanding [CBB+02, BCK03]. Understanding the architecture is for instance the key to understand performance and reliability.

What kind of documentation is needed for newcomers to open source projects?
One study in [MFH02] presents the example of the Mozilla project, where documentation may have been a contributing factor for the lack of outside contribution in its first years. Complex software developed with open source may need to take measures in order to reduce barriers for newcomers to contribute.

Properties of architecture that are important for both adopters and developers are i.e. design patterns, rationales for design choices, performance and scalability of modules, and API compatibility and adherence to standards. Documentation may also serve as a vehicle for learning for various types of stakeholders.

Maintaining consistent architecture in open source projects are difficult for specific reasons. Commercial development rarely has significant drift between architectural plans and their implementations, while the contrary generally is true in open source due to its distributive nature [PC04]. A cycle of reinvention [Sca04] that is typical for many open

source projects may be the cause.

Participation to an open source project may be helpful to highlight the difficulties in participating due to the existence or non-existence of documentation. Open source projects are known to generally provide little documentation [VV04]. It is interesting to look at reasons for this tendency, and the extent to which the assertion is true in the Apache Derby project.

1.4 Overview

Chapter 2 presents the research method used in this project, the time schedule for various activities, and data collection. Chapter 3 presents open source in greater detail. The database system Apache Derby is then presented in Chapter 4, which is the instance of an open source project that this research primarily will focus on. Chapter 5 will present the state-of-the-art of documentation in open source communities.

Activities in this project that relates to the participation to the Apache Derby project is presented in Chapter 6, followed by a discussion of the research questions in Chapter 7, and conclusions in Chapter 8.

Chapter 2

Research method

Using a rigorous research method is important in order to ensure that results will follow from the premises of the research situation. Work in this thesis is based on Canonical Action Research [DMK04].

This chapter will further explain the Action Research method, and how this research method will help to accomplish the goal of this thesis.

2.1 Action research

The role of a research method is to provide a basis for interpreting the world, and serve as a framework to collect and analyze data. Methods consist of a series of rules or guidelines to ensure that conclusions follow from the premises.

Action research is a research method that is iterative and collaborative in nature. The brand of action research used here, Canonical Action Research (CAR), is based on the principles of working with an organization, and observe an organizational change process. The problem being examined is understood through interacting with or changing the organization through well-planned actions. It is similar to ethnographic methods, such as participant observation, in that it studies the phenomenon in the context it naturally appears in.

The iterative nature of Canonical Action Research is intended to provide a systematic approach to the problem. Having well-defined phases ensures that the understanding of the problem may be evolved in several stages, and that success criteria of the research may be properly addressed. Iterating towards a goal with planning, execution and evaluation phases also allows for brief pauses where results of the research may be communicated and discussed.

In organizations, information is provided at different levels. Information may be known privately or collectively, implicitly or explicitly [Dic93]. This relationship is illustrated in figure 2.1. Being a participant in the organization is then helpful to gain insight into the practices used by the members. Rationales for practices and their benefits and drawbacks can then be documented from an outside perspective, from a researcher that

is part of the system, but remains committed to the research goals.

	Tacit	Explicit
Individual	**Experience** Skills and know-how	**Conceptual knowledge** Symbols and concepts
Collective	**Routines** Daily operations, culture	**Systemized knowledge** Documents, databases

Figure 2.1: Knowledge assets: Tacit and explicit knowledge (Nonaka and Takeuchi)

In comparison to Grounded Theory [Pan96] research, which reverses the order of hypothesis generation and data collection, action research focuses on the evolution of a theory or understanding, or even evolution of research questions. Also comparing to classical research, action research may be an "upstream" method, where creativity and analytic skills of the researcher are vital for reaching the goals [Dic93]. The CAR process also depends on the researcher's abilities for thorough interpretations of the situation, the analysis of causes and effects, or proper communication of the facts and their interpretations.

The technical skills and ability the researcher has to participate to and change the organization is helpful, but not the only factor determining the success of the intervention.

2.1.1 Researcher-client agreement

One of the prerequisites for using Action Research is that the researcher is able to be part of the studied organization. Being able to observe an organizational change process involves being able to observe outcomes, and interact with it.

The researcher should have some sort of agreement with the organization, or better yet, a written agreement on the expectations and goals of the research. For research in open source, where consensus is the main control variable, this agreement can take the form of an introduction about the researcher's intentions, or may be dropped altogether if it is found ethically and functionally feasible. An agreement will for open source, in the understanding of this thesis, be the social contract in which work towards a common goal is done.

2.1.2 Cyclical process model

The cyclical process model of CAR consists of the following phrases, as found in [DMK04]:

1. **Diagnosis**: The purpose of the diagnosis phase is continuosly making sure the activities in the research are relevant to the problem being examined. In an action research project that spans several iterations, the diagnosis will take the current situation understanding and integrate it into the research agenda. For instance, research questions may be further refined or elaborated upon.

2. **Planning**: As CAR involves participating to an organizational system, the planning of actions should be done carefully so the correct data can be collected. The planned actions need to be accompanied with an assessment of what the expected outcome will be. The planning phase is informed directly by the Diagnosis phase.

3. **Intervention**: CAR interventions may be restricted to days, or even months in length. The planned actions are implemented as they are planned. Data collection should collect all the relevant observations from the intervention, but also thoughts and reflections during planning and evaluation phases.

4. **Evaluation**: The results from the intervention should be subsequently analyzed with regard to the expected output, and what has been learned. The evaluation phase should be performed to determine whether the intervention was implemented as planned, in order to document what has been learned, and if any cause/effect relationships are correct.

5. **Reflection**: The researcher should reflect upon his own actions for the purpose of learning from them. Outcomes from the project should be evaluated in context of the project goals. If project goals have been accomplished, then a decision should be made to exit the project.

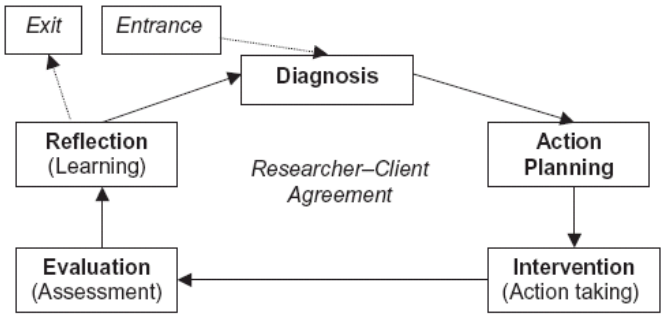

Figure 2.2: The Canonical Action Research cyclical process model, [DMK04]

Figure 2.2 shows a typical action research execution. The strength of the CAR approach is that it responds to the continually changing environment. The researcher grows a better understanding of the situation, while providing suggestions or analyzes that are relevant for practitioners.

2.1.3 Theory and reflection

Theory and reflection have critical roles in action research. Action research addresses a problematic situation in an organization, combining theory and practice through change and reflection in order to contribute knowledge within and outside the confines of the project [DMK04]. Action research without theory is simply *action learning*, and only regarded as research in a positivist tradition.

The reflection principle of action research allows learning through understanding organizational norms, and *"the advancement of knowledge by generating new theory or informing/re-informing existing theory."* [DMK04]. This demonstrates that theory and reflection are in fact related, and that action research can extend existing knowledge in a scholarly tradition.

2.2 Research and education in open source

Open source has by definition some degree of transparency to the outside world. Open source communities are usually open for contributions and observers from the outside, which makes them good candidates for empirical research. The transparency allows assessment of the quality of open source projects by evaluating their activity levels and popularity. These are in many ways conductive to the level of innovation, and ability to provide corrections to the software. For open source projects that are alive and undergoing active development, it is simpler to participate in a research situation.

Research in open source allows a student to learn both scientific methods, technical skills, while contributing to scholarly knowledge. Action research allows the student to start with basic understanding gained from literature, and consciously reflect on her or his learning. Observing how software development is done in practice further allows the student to reflect on how the software engineering profession is implemented in practice.

Research that aims to provide practical guidelines should concentrate on organizational problems and challenges in the profession. Methods to provide relevance in research include selecting a topic that is directly relevant for practitioners, basing the research on context-rich theories and a cumulative research tradition, and portraying the research output so that it can be used to rationalize decisions in an organization [BZ99]. In addition to the relevance of research, the rigor[1] is also important especially for reviewers of research.

This thesis is based on using action research, and aims to extract knowledge about open source through participation to one open source project. The relevance of this research for practitioners will be in that it provides suggestions for improving open source practices, or may represent a useful case study for implementers of open source projects.

[1] Rigor means "Strict 'precision' and 'exactness'," [DMK04], or "the correct use of methods and analyses appropriate to the tasks at hand" [BZ99]

2.3 Previous action research experience

There have earlier been studies involving the use of action research to study open source. A master's thesis in 2003 uses action research to study acceptance and integration of newcomers in small open source projects [TT03]. More recently, the undersigned did a directed study in the Netbeans community in autumn 2005, looking for effects of commercial maintainers of open source communities [Mor05]. Using action research and open source in the context of education is further explored in the article in Appendix B.

A number of lessons were drawn from the directed study reported on in [Mor05]. For instance, it was found that selection of an open source project to study should also consider the researcher's abilities and interests. If the research design is dependent on contributing to the project, it imposes demands to the researcher on technical proficiency and interest in the project's goals.

Some of the issues that action research itself introduces, which should be expected in such a project, included:

- **Entering**: What does it mean to enter an open source project? CAR literature has a *Researcher-Client agreement* as one of its principles. Such an agreement is difficult in decentralized open source projects, and poses ethical challenges such as whether or not the researcher should introduce himself to the community as a researcher.
- **Collaboration**: The researcher should attempt to discuss findings with other participants or peers in order to understand and evolve theories. Multiple viewpoints eases the critical interpretation of findings. Further asserted by Davison et al. in [DMK04]: *"[t]he researcher must account for the values, beliefs and intentions of the client employees, and treat them as collaborators rather than mere research objects"*. This may imply that community members should be introduced to findings, or it may involve investigation of historic communication on the topics of interest, such as mailing list archives.
- **Learning through reflection**: Explicitly stating what has been learnt is the most critical activity of CAR [DMK04]. A requirement for the reflection to be efficient, the researcher should have performed a thorough and systematic literature review in advance. This shouldn't be neglected unless the goal focuses on learning alone. The researcher should in advance have thought about which results can be anticipated, the possible consequences of these, and how to prepare for the consequences.

2.4 Data collection strategy

Using action research in the context of open source research imposes constraints on which data is available for collection. The cyclical process model of action research requires that actions are planned ahead of each research iteration. The evolving knowledge of the studied topic will thus influence what data is collected.

Research questions are the primary guides for which activities that will be undertaken. The activities should in accordance to CAR be directed towards answering the problem at the current level of knowledge. What types of data that may be useful for future analysis include, but is not limited to:

- Descriptions of project artifacts, their design principles, and overall functionality of them, which constitute the frame of the study.
- What kind of documentation exist, in what form, how updated it is, thoroughness, availability, and communication relating to it.
- Any practices related to architecture descriptions and documentation. How important is documentation regarded ("culture"), and how important are open standards in this regard.
- What kinds of problems are frequently discussed in the community, what are the developers worried about?
- A journal is maintained with all activities, the progress of tasks, and information on the outcomes of action research interventions.

The collected data will be in the form of a log of activities, rationales for the activities, and their outcomes. An attempt will be made to collect as much relevant information as possible. This will contribute to enhance the understanding of the problem, and subsequent qualitative analysis may help render the observations into knowledge.

2.5 Data analysis strategy

The data collected during the participation phase will be analyzed to find theoretical interpretations in the context of the research questions. This research is based on observing how an open source community works both on the technical and community level, which also involves understanding the social dimension. Research based on participation is appropriate to generate theoretical interpretations [Jor89], but may be less suited for testing hypotheses.

It is interesting to compare findings from open source to more elaborate software development processes that typically are used commercially.

Methods employed to analyze the data will be the following:

- Compare the interaction journal with other sources, like Apache Derby's web pages, other literature, and written communication of developers. This is to try to address the one-sidedness of focus, and reduce the possibility of misconceptions.
- Findings can be evaluated according to smaller dimensions of the problem. Dividing the problems into subproblems will reduce complexity, and allow more intuitive analyzes.

- A theorization process will be based on intuition, and thinking aids. Other literature can be studied to create alternative viewpoints to the problems, but literature should for the sake of rigor not serve as explanations. This is a creative process.

Chapter 3

Open source

The purpose of this chapter is to provide a brief overview of open source. When discussing this concept, it should be recognized that there are no single ideal implementation of the open source principles. Development processes are not strictly enforced, and are often particular to the system being developed. For instance, the FreeBSD project has a defined process for dealing with releases [1].

Basic principles of open source requires that software must be freely redistributed to any interested party. Ownership of the code is still based on who wrote the original code, but through applying an open source license to computer code, the right is given to anyone to make derivative versions of it. Open source is light-weight in its process requirements, but requires implementations to emphasis code sharing, and transparency.

3.1 Definition

The official definition of open source is controlled by the Open Source Initiative (OSI) [Ini05], which has trademarked the term in order to protect its meaning. The definition concerns the distribution terms of open source software, essentially their license agreements, and amongst others requires that source code is distributed together with any copies of the software.

For a software development project to be open source, Bleek and Finck presents three criteria they regard as essential for it to be called open source [BF04]:

1. **Openness**: The project must allow new developers, and that anyone can use the product.

2. **Agility**: The development must follow short cycles, and the process may be easily changed.

3. **Distributed**: The developers are not located physically at the same place.

[1] http://www.freebsd.org/doc/en_US.ISO8859-1/articles/releng/release-proc.html

These requirements are compatible with the open source definition.

Scacchi presents in [Sca04] a comparison of phases in modern software development processes and its corresponding mechanisms in open source. His findings are used to try to create an overview of how open source communities actually produce software. Requirements elicitation is found to be a by-product of a community discourse on what the product should or should not do. Maintenance of software is based on evolutionary redevelopment. Furthermore, a larger community will have the opportunity to maintain an increasingly complex system, reflecting on a co-evolution of product and community.

3.1.1 Communities

An open source project is a community of people who share interests, and work together in order to solve a common itch. The individuals that participate may have various motives for participating, but the individuals motives are hardly ever questioned. One typical motivation for organizations engaging in open source development is the need for some particular type of software for in-house use, i.e. middleware for offering on-demand web services.

The development communities may be characterized as virtual. Participants are usually spread out across various time zones, and communicate typically only through virtual, open forums on the Internet. These virtual teams have inherited cultural and organizational schemes from the scientific community, but progressively enriched those to fit open source development processes [Maa04].

People participating in such development communities can be said to enter various self-assigned roles, based on which interest they have in the project, their skills and desires to improve the product, and how much time they are willing to invest. Most people affiliated with an open source project will be reporting problem, while a significantly smaller group will be working on fixing defects or implementing new functionality [MFH02].

3.1.2 Management styles

Also pointed out by Scacchi in [Sca04] is the differences in management styles. Leadership usually takes the form of an interlinked and layered meritocracy. People with merits can get roles as committers, meaning they have access to change the project artifacts stored in a central repository. Communities like the Apache Software Foundation (ASF) also embrace the roles enabled by a meritocratic leadership [Fou99]. ASF also operates with the notion of "do-ocracy" – *the power of those who do*, to recognize the decision making power given to people that volunteer to get a task done.

Implementations of open source development processes can have different views on how open source communities are lead. Projects managed under the Apache charter makes important decisions through voting. Votes are either `+1`, a positive vote, `0`, meaning no opinion, or `-1` [Fou99], the latter which in essence is a veto to the proposal. This

is referred to as the *lazy consensus approach*. Other open source processes may impose less strict management, such as all with CVS/SVN[2] write access can make decisions from their discretion. Another instance is the Mozilla.org project, where project management related to specific releases are managed by *drivers* [Moz06], who on a patch-by-patch basis decides what is included.

Other open source projects can have more rigid leadership styles, like the Linux kernel development lead by Linus Torvalds, where one dictator makes decisions on acceptance or rejection of code changes [Sen04].

It is not obvious that any software development will work with open source. Using open source has benefits for attracting free testing and free development help, but only if there are incentives for people to contribute. Jamie Zawinski, the original founder of the Mozilla.org web site, observed in 1999 as he resigned from Netscape Communications and Mozilla.org work, that *"you can't take a dying project, sprinkle it with the magic pixie dust of 'open source,' and have everything magically work out."* [Zaw99].

3.2 History

The tradition of sharing source code is one that has been around since it was in the laboratory stage at universities. Some notable communities of so-called *hackers* existed in the 1960s and 1970s in the USA [Ras00], where sharing of code was as natural as how scientific communities today share knowledge.

The notion of freely accessible source code were in the following decades going to be challenged by an increasing commercialization of the computer science field. Bright minds were hired by large corporations, and the hacker culture was in decline. The seminal writing of Richard M. Stallman illustrates this process, as well as his work to promote free software through his very own GNU Project [Sta99].

One model of software development that is closely related to open source is the *free software* movement. The open source versus free software terminology has long been a source for confusion for many. While they in most aspects are identical, there are some open source projects that would not fit the ideas of the free software movement. The biggest problem with the free software concept has been the word *free*, which in this context is not synonymous with being free of charge, but rather the freedom that opposes to secrecy of computer code. Free software also have other reservations against being commercially exploited.

Some proponents of free software now accept the open source term also to cover free software, in order to avoid confusion, i.e. in [Ray01]. Idealization of the open source movement is however usual, presenting open source software developers as *hackers* fighting for software freedom, being people that are naturally anti-authoritarian (ibid):

Hackers are naturally anti-authoritarian. Anyone who can give you orders can

[2]CVS and SVN are two popular versioning systems, that allows multiple versions and changes to them to be represented.

stop you from solving whatever problem you're being fascinated by – and, given the way authoritarian minds work, will generally find some appallingly stupid reason to do so. So the authoritarian attitude has to be fought wherever you find it, lest it smother you and other hackers.

3.3 Principles

Eric S. Raymond's treatise from 1997, *The cathedral and the Bazaar* [Ray00] presents interesting examples on advantages of an open source development model, and how hacker culture fits into this picture. Here, Raymond presents advice for open source development through examples. His advice include i.e. *"Release early. Release often. And listen to your customers."*, and *"Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected."*

No matter how one looks at the free/open source phenomenon, it is a complex social discipline with several success stories. The Internet is built on free software. More recently, notable desktop applications such as OpenOffice have become famous for being freely available. The web browser Mozilla Firefox is the product of a full remake of the older Netscape browser [Moz06].

The recent success of open source has often been attributed to the "many eyeballs" nature of open source development. Developers of open source software benefits from a large community that provides feedback and bug reports, and through that improving the product. The success from the debugging perspective can be attributed to the activities developers undertake for reporting, understanding and resolving bug reports [Østerlie06].

3.4 Licenses

Software licenses are the reason why open source is a viable system. The gift culture which open source is based on, relies on that developers and companies are willing to share their work. In order to prevent freeloaders to utilize volunteer effort to make money, there are restrictions on combining purely commercial and open source code.

Perhaps the most used license in the open source world is the GNU General Public License (GPL)[3]. Software authors applying this license to their programs gives users the right to redistribute the software freely, as long as the code and license terms follow it. The copyright to the source code remains with the author, but anyone can potentially make alterations to the program and release it under a new name. However, this also requires that the GPL follows it.

The incentives firms have for participating in open source are by Henkel found to be based on license requirements, the company's reputation, and getting bugfixes and

[3] `http://www.gnu.org/copyleft/gpl.html`

maintenance for no additional cost [Hen05]. However, following open source licensing schemes have the disadvantage of potentially strengthening a competitor's position in the market: *"Firms can and should balance revealing and protection in such a way as to optimize their pattern of free revealing."* [Hen05]

Chapter 4

Apache Derby

The research in this thesis is focused on participation to the Java database system Apache Derby. This chapter will give an overview of the main functions of this system, its architecture, and how the development of the system is done.

4.1 History

The history of Apache Derby starts in September 1996 with a small start-up company in Oakland, California, namely Cloudscape, Inc., who developed a Java database system dubbed JBMS. JBMS, which later was renamed Cloudscape, was then an embeddable database engine much like its descendant is today. It supported a small subset of the SQL-92 query language standard. Some of the features used to sell it was its small footprint, zero administration, and embeddable qualities.

Cloudscape was developed years before Java had become widespread. In 1999, Cloudscape Inc. was purchased by Informix, another database vendor. In 2001, IBM purchased the database assets of Informix, which also included the Cloudscape database system. IBM continued to develop the product in addition to its new Informix database, and its flagship database product IBM DB2.

The more recent history starts on August 3, 2004, when IBM released version 10 of Cloudscape, and simultaneously released the product with an open source license. The Apache Software Foundation subsequently accepted the system as one of its projects, which now became Apache Derby. Today, Apache Derby is supported by major corporations interested in its development, mainly IBM and Sun Microsystems. This support takes the form of contributions to the Apache Derby code, funding developers working on the project, and providing hardware resources for the development [ZSB05]. Products like IBM Cloudscape and Sun Java DB are released as derivatives of the Apache Derby code, and their respective companies sell support services for them.

Apache Derby is currently being maintained with the collaborative framework that open source and the Apache Software Foundation provides.

4.2 Standards compliance

Innovation has long been one of the acclaimed hallmarks of open source. Innovation is the result of hackers getting free leashes to implement solutions as they prefer them to be. However, in order to reduce the number and scope of implementations, open standards and standard compliance also has a significant role. Standards allow multiple implementations of the same concept [Wal06], and also allows related products to talk with each other.

Apache Derby is in this regard not different. Participants of the Apache Derby project value standards, and decisions are often based on their adherence to them. Apache Derby is based on two main standards: That of JDBC (Java Database Connectivity), and that of SQL (Structured Query Language).

4.2.1 Structured Query Language

Perhaps the most widely used querying language in use is SQL, which is used to retrieve information from relational database systems. The SQL language has undergone various revisions, from the original standard in 1986 (by ANSI) until 2003. Features such as identity columns and triggers are added in the more recent revisions. An example use of SQL is shown in figure 4.1.

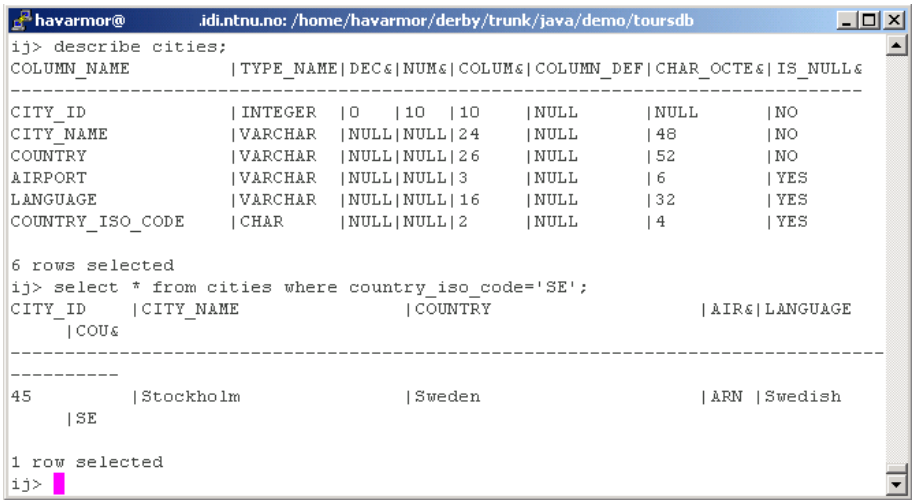

Figure 4.1: Example SQL query in Derby's command line tool, IJ.

The purpose of SQL is basically to manage data or meta data in a centrally stored database. Having a standardized language to retrieve this information allows for database systems to be replaced with little or no changes in the query specifications.

There are three major types of queries. *Select* statements are used to retrieve information from the database in the form of a set of matching rows. *Data Definition Language*

(DDL) statements are used to change the structure of the stored data, specify constraints on it, or specify the layout of the database itself. The last type, *Update* statements, allow for changing the stored data.

4.2.2 Java Database Connectivity

Java Database Connectivity (JDBC) is a set of APIs, or interfaces, that define the behavior and operation of a JDBC-compatible database system [Mic05]. The API may contain functionality such as executing queries against the database, retrieving meta information, transaction management, and handling of various SQL data types. JDBC API allows for SQL queries to be executed on the database, without the programmer needing to know detailed inner workings of the database system that is being queried.

Another advantage of the architecture of the JDBC API is that it does not make design decisions on the system. Database systems may be embedded in the same run-time environment as the application, or may be operated as a network server over the Internet. It may as well be implemented in a different language or platform. Figure 4.2 shows an application's relation to Apache Derby, and how JDBC is used for communication (JDBC includes also the ResultSet and Statement interfaces).

While SQL queries are being fed to the database system unchanged, the resulting data from queries are embedded in ResultSet objects. Result sets are objects that allow the caller to fetch any rows that match the query. The result sets represent cursors to the database data on the server, and rows are returned on demand.

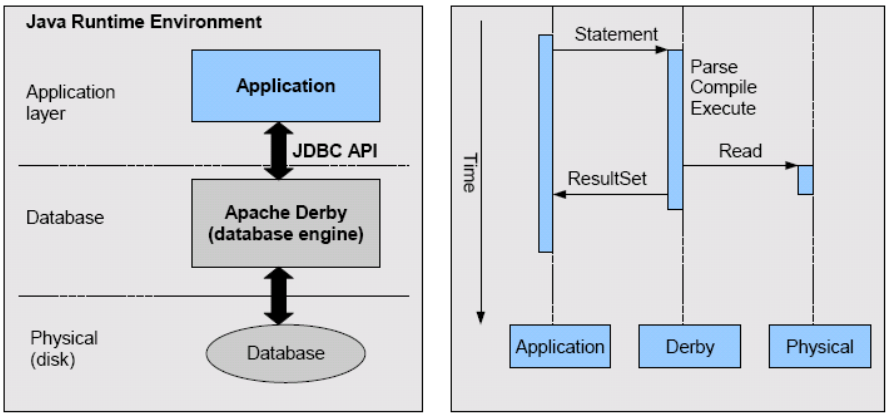

Figure 4.2: Conceptual module diagram (layered view) and sequence diagram for application/Derby relation to JDBC.

The JDBC standard also undergoes changes. At the time of writing, JDBC 4.0 is currently under review, which is a minor revision of previous JDBC versions. The binary compatibility between versions is important. Users of the Java platform can have widely differing versions of the Java platform installed, thus one of the issues of new versions

of standards is to ensure that older applications still works on newer Java Runtime Environments.

As JDBC depends on the SQL standard as a query language, the standard does not in itself decide how a database system operates. For instance, Microsoft's implementation of SQL used in products such as SQL Server 2000, Transact-SQL, has a fundamentally different method for accessing database meta data than SQL-J, which is used in Apache Derby. SQL-J relies on JDBC's implementation of DatabaseMetaData for this purpose. Differences like these, however, often have annoying consequences like the need to create several implementations of a database access mechanism for applications that should operate with different database systems.

Some of the important elements of JDBC are:

Driver The driver interface typically will represent a database system, or an access method to a database. Database systems will need to implement the driver, and applications need to instantiate them when database connectivity is needed. Apache Derby has two drivers, namely `EmbeddedDriver` and `ClientDriver`, that represent running Apache Derby as part of the application, and running it as a stand-alone network server respectively.

DriverManager The driver manager is the Java entity that loads JDBC drivers. The loading of drivers are based on URLs that specify which database system and database to connect to, including properties such as whether the database should be encrypted, user name, passwords, etc.

Statement Statements are used for executing SQL statements. The Statement interface distinguishes between the execution of statements returning data, and those updating data or meta data. A special case of the Statement interface is `PreparedStatement`, which is a pre-compiled SQL statement that allows JDBC database systems to optimize repeated similar operations. Apache Derby optimizes statements by generating executable Java byte code for the SQL, which the Java Runtime Environment may further optimize into platform-dependent byte code when repeated calls are made.

ResultSet Result sets are tables of data that represents the result from a query on the database. The result set will allow the application to fetch rows, or tuples, from the database in the order specified by the query description. Apache Derby provides several implementations of result sets for internal use, which serves the purpose of handling functions such as sorting, index look-ups, and table scans.

Types JDBC stores information on the data types of columns as part of its meta data. The data types of JDBC are similar to those of SQL, and are referred to as standardized names such as INTEGER, FLOAT, DATE, and VARCHAR. The implementations of the data

types in Apache Derby are all based on a generic SQL data type, so they are able to offer a type conversion and comparison operations.

DatabaseMetaData, ResultSetMetaData The meta data classes allows for an application to determine the structure of the data in a given table name, a view, what SQL procedures are accessible, or for result sets which columns are included in the result set.

4.3 Features

In addition to being a database system implemented in pure Java, Apache Derby also advertise having good performance in comparison to its siblings implemented in native languages. Apache Derby also is based on the ACID[1] principle, which means that it ensures the correct execution of transactions.

Embedded As already mentioned, one of the most useful features of Apache Derby is the possibility to bundle it with any application, without the application being required to follow the same license. This conveniently allows the database to be combined with open source middleware systems such as Hibernate and JBoss.

Tool support Tools such as `ij` and `dblook` are JDBC-compliant tools that can be used with Apache Derby. `ij` is a scripting tool for automating operations related to testing, and creation/maintenance of databases. SQL-J queries are taken as inputs, and the resulting data from the queries are displayed.

4.4 Community and infrastructure

The Apache Derby community is composed of the persons participating to the project, and supported by a network of communication mediums. These mediums most importantly involve storage for source code (a SVN repository), an issue tracking system, and mailing lists. There are also various web pages that are useful for users.

These infrastructure services are being offered by the Apache Software Foundation (ASF), which is a non-profit organization formed in 1999 to provide a technical and legal foundation for *"open, collaborative software development projects"* [Fou99].

Projects operating under ASF's umbrella have full autonomy, run by consensus of its members, but an *officer* of the ASF needs to be present in each project's Project Management Committee (PMC). The role of the PMC is to ensure the health of the community, and that development happens in an orderly fashion. Furthermore, the presence of an ASF-appointed officer in the individual projects is a key to litigation protection, as the foundation as a whole will be implicated in case of any lawsuits.

[1] Atomicity, Consistency, Isolation, and Durability (of database transactions)

4.5 Architecture

The logical architecture of a software system is the structures which the software is built on. It is independent of technology. This differs from the physical architecture, which is the set of modules and classes that communicate together. This section briefly shows the principles of Derby's architecture. A conceptual sketch is shown in figure 4.3.

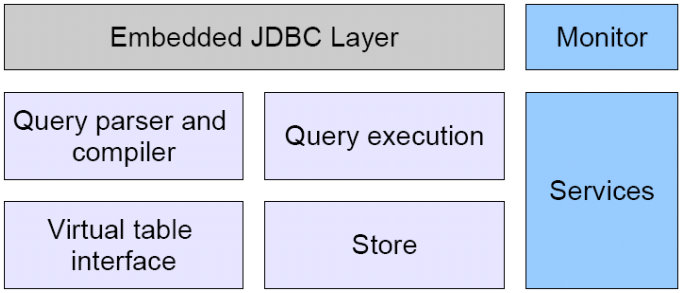

Figure 4.3: Logical architecture of Derby, from [Deb04]

Monitor The monitor[2] is the object that loads and instantiates services and modules (explained below, and shown in figure 4.4) in the system. When i.e. a new database should be loaded, the request is handled by the monitor.

Service Services are sets of well-defined functionality used in the system. A database is in itself a service. They consist of one or more modules, where one of the modules must be the *factory* class for the service.

Modules Modules are isolated parts of functionality, such as a lock manager, or data access/indexing method. Modules can be either embedded in a service, or they can be system-wide, such as error logging.

Query processing The processing of SQL queries is a multi-phase process which proceeds from the textual input of the query string, to the return of data and state from the database. Important for the query processing are factors such as standard compliant behavior of the API, the reliability, and the performance at which data can be returned.

Data retrieval queries go through the following phases:

- **Parsing**: The query string is parsed with a JavaCC parser, which traverses the query from left to right, generating a hierarchy of nodes of the input. Constant values are also bound to their respective query tree nodes.

[2] `http://db.apache.org/derby/javadoc/engine/org/apache/derby/iapi/services/monitor/Monitor.html`

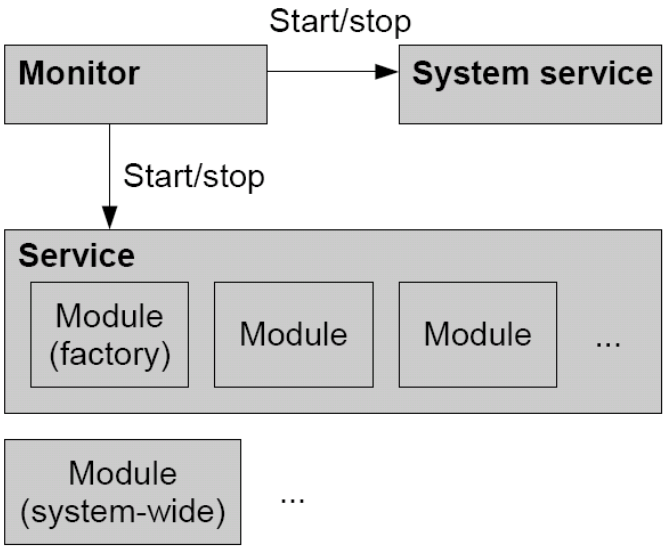

Figure 4.4: Relationship between services, modules and the monitor.

- **Optimization**: Preprocessing can eliminate unnecessary constructs, or optimize queries by prioritizing more selective constraints to reduce the number of comparisons. In cases where the entire query is not known, such as when using pre-compiled `PreparedStatements`, query optimization is often not possible.
- **Compilation**: The query tree nodes are used to *generate* runnable Java byte code for the query. The purpose of this is to optimize the execution and comparisons of queries. Especially when using pre-compiled SQL statements, the performance benefits of this step are large. The generated code is based on a `Activation` interface, and uses public Derby API for data type access, comparisons, database store access etc.
- **Execution**: Execution of queries involve running the compiled query. A hierarchy of result sets are generated by the execution. The returned set of matching rows may be piped through i.e. both an index scan result set, and sorting result sets. This allow Derby to efficiently manage memory, while avoiding a very complex query processor. The result will be presented to the user in a `ResultSet` object, which allows for retrieval of one row at the time.

Figure 4.5 shows how the query processing works in six stages, starting with the original query and ending with returned data. The purpose of this illustration is to demonstrate the complexity of the query processing pipeline.

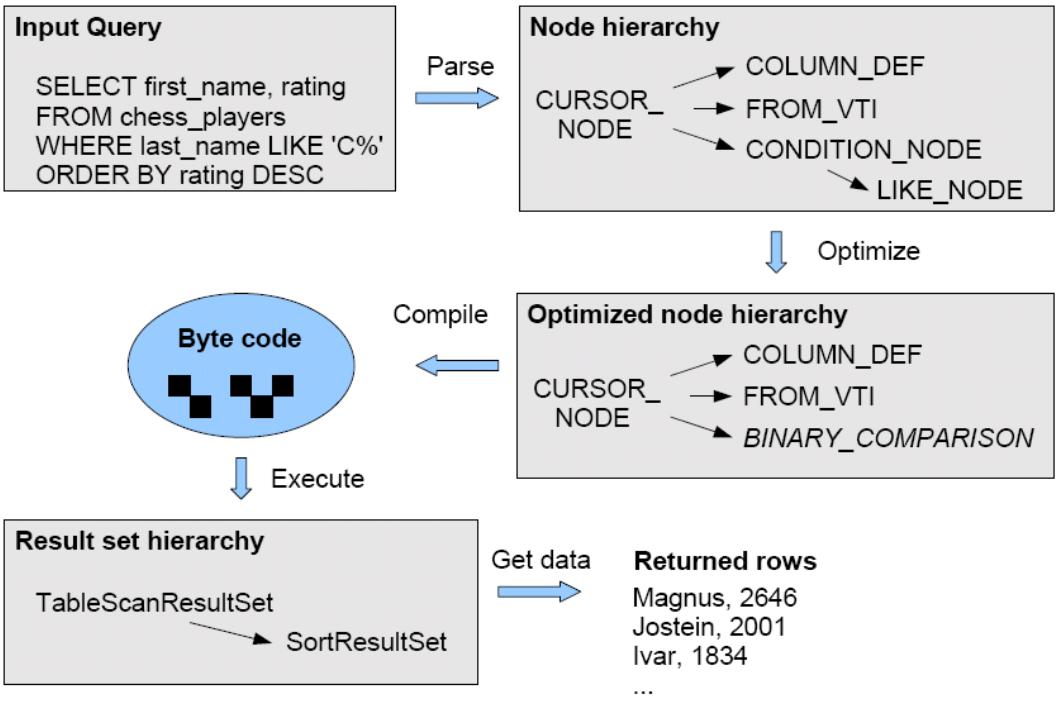

Figure 4.5: Overview of parsing and execution of SQL queries in Apache Derby (note: node hierarchies only examples, not necessarily factual).

Chapter 5

Software documentation

The common perception of software documentation is that it is written for users, in order to make the software easier to understand. It enables the users to accomplish their goals for using the software. User documentation is not addressed in this study, but it is worth pointing out the distinction.

Software documentation is by [For02] described as *"an artefact whose purpose is to communicate information about the software system to which it belongs"*. This definition encompasses documentation for users and developers, and equates it with communication.

The documentation is supposed to demonstrate parts of the software that is not immediately understandable, while leaving out the internals that does not help the understanding of the reader. For developer documentation, or architecture descriptions, it can typically be used to simplify the task of doing maintenance operations on the source code, or integration with other systems.

As an effect of the status-driven, volunteer culture of open source, any kind of documentation in open source processes is traditionally the last element to be developed. That is, if it is ever developed. Raymond describes this problem in [Ray00]:

> *It is a hallowed given that programmers hate documenting; how is it, then, that Linux hackers generate so much documentation? Evidently Linux's free market in egoboo works better to produce virtuous, other-directed behavior than the massively-funded documentation shops of commercial software producers.*

This chapter explores the idiosyncrasies of architectural documentation, and its various forms as used in the software engineering discipline.

5.1 Software architecture

Software architecture is by Bass et al. defined as follows [BCK03]:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

In addition to the structure or structures of a program or computing system, it is also important to have a method for representing these in a manner that is understandable for humans. The software architecture discipline accomplishes this by reducing the complexity through looking at the software through various viewpoints. Distinguishing between high-level perspectives of the system and lower-level detailed architecture also helps reduce complexity for a reader.

Figure 5.1 shows an example of a software architecture. The description shows which parts of the system communicate. A full description would typically detail each of the modules in greater detail, and give rationales for design considerations.

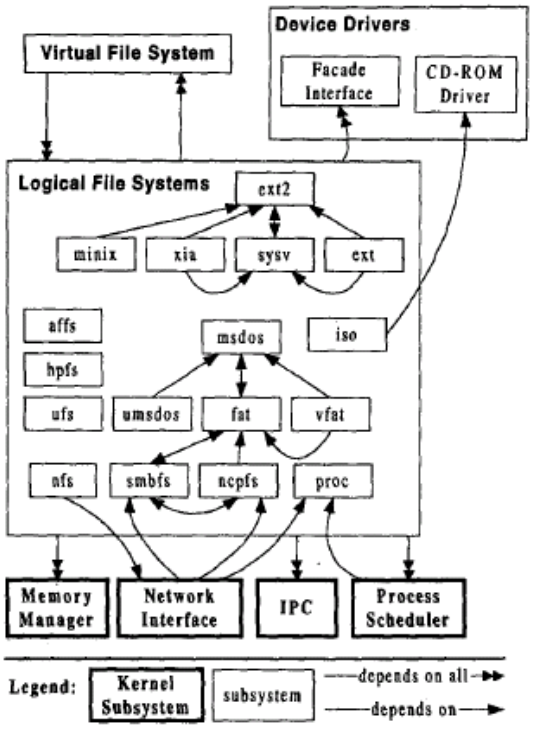

Figure 5.1: Example of a logical architectural description, the file system in the Linux kernel, from [BHB99]

In a system's lifetime, the software architecture is continuously evolving and changing with the new requirements. New technologies emerge, and infrastructure supporting software need to be adapted to the new environment. The software documentation here plays the role of a human-readable representation of the architecture, which allows developers to identify design principles and potential modifications made necessary by a new requirement.

Using a documented software architecture throughout the lifetime of a software system can improve the quality and maintainability of the system [BHB99]. The consequence of not having a documented architecture are higher maintenance costs, and potential problems with consistency in the software.

5.1.1 Software design

An important distinction to make is that of between design decisions, and software architecture. Software architecture deals with externally visible elements of the software elements. Elements that have no interest outside of its module are not considered subjects of architectural decisions. How elements are implemented, or which technology they use to accomplish their goals, should be up to the implementer to decide.

The meaning of this is that architecture is a high-level abstraction of the system. This addresses the distinction between conceptual and concrete architecture [BHB99]. A software architecture can describe the ideas behind the system's design. However, the concrete architecture may differ because of performance optimizations, and other design decisions.

5.1.2 Views

One proposal to simplify the organization of software architecture documentation is the 4+1 view model by Kruchten [Kru95]. The purpose of the 4+1 view model is to organize descriptions of architectures in five concurrent views, which isolates concerns. The views are the following, as described by [Kru95]:

- **Logical** The logical view describes the classes in the system, or the relevant entities that work together. This description can be a class diagram, or i.e. an Entity-Relationship diagram.
- **Process** The process view shows concurrent processes or threads, and how these are synchronized to each other.
- **Physical** The physical view shows how the software is deployed to hardware, and the distributed nature of any components.
- **Development** The development view describes how the software is developed by the organization. The various personnel working on the software is relevant. This is in accord with Conway's law, which suggests that the organization of a software system will be congruent to the organization of the team that designed the system [BH99].

In addition to these four viewpoints, Kruchten also recommends having a set of usage scenarios, or use cases, that describe the intended use of the system. These views are illustrated in figure 5.2, together with which stakeholders and concerns are important for each of the dimensions.

Each view in a documented software architecture are in themselves not sufficient to understand the system, but together enough to understand how the system works. Kruchten also argues the need for a design document should accopagny the software architecture, to explain lower-level design decisions that are needed to understand the system throughout its lifetime.

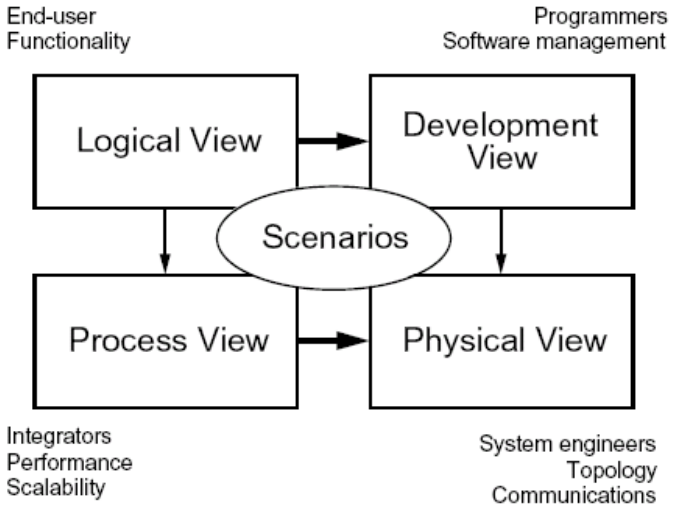

Figure 5.2: Kruchten's 4+1 view model, [DMK04]

The people that are affected by the software architecture process or the resulting system are referred to as *stakeholders*. This title implies that they have interest in how the system is developed, like the cost and end-user support requirements, or how the completed product affects their job.

5.2 Issues with documentation

Smith et al. in [STT01] points out the confusion surrounding documentation of viewpoints of the software. Little is known about which type of documentation is useful, how it should be written, while also questioning motives for having documentation at all. This is by Smith et al. exemplified by light-weight processes like Extreme Programming (XP), where documentation is done away with completely.

Criticism of the agile software development processes like XP include that it can border to the irresponsible with regard to formality, and an undue reliance on individual competency, as cited in [Koc04]. These problems may also be present in open source.

The IEEE Recommended Practice for Architectural Descriptions positions the architecture of a system as an activity which has the purpose of reducing the formidable risks and difficulties of design, construction, deployment, and evolution of software-intensive systems [IEE00]. This reflects on a consensus among IT professionals that architectural

descriptions should be made. It does not, however, distinguish between technologies or development processes used for creating the software. In other words, properties of the open source development process may in itself be enough to document architecture.

5.3 Open source relation

It is widely recognized that large software systems should have a documented software architecture [Kru95, BHB99]. This assertion is easily understood in the context of commercial software development. However, for open source development processes, this brings some problems.

The production of user documentation in open source has historically known to be insufficient for many uses, such as described by Richard Stallman in [Sta99]. Here, it is asserted that the development of documentation should be parallel to the software, and encourage its maintenance by offering the same freedom to modify it, as open source software allows.

In much contemporary open source literature, the issue of software architecture documentation is ignored. such as in [BP01]. A reason for this may be that open source development processes traditionally have no requirements for documenting software architecture, and it is simply not given a thought. Commercial software development uses architectural documentation in order to attain high maintainability and overall quality [BHB99]. The same effect is attained in open source through its highly distributed nature, and access to enormous availability of free labor force. There are not many incentives for the average open source project to be trying to close the gap to commercial development with regard to architecture documentation.

One situation where the current open source practices on documenting software architectures is insufficient, is when it is used in narrow, specialized domains where there are fewer interested developers. If an open source process is to be used here, then better consideration needs to be given to the documentation aspects. This is however a non-issue in the software engineering community.

Bowman et al. [BH99] has suggested that analyzing the relationship between developers and code can be visualized and used to reconstruct the architecture. This is not unlike existing open source practices, where bug tracking systems and mailing list provide vital clues to who-knows-what.

Chapter 6

Participation

This chapter presents the research activities done in this project, and the evolution of the theory. The purpose is to allow reviewers to see how the results are found. Section 6.1 gives an overview of the participation phases and the philosophy behind their planning. Sections 6.2, 6.3 and 6.4 shows the processes for the three executed action research iterations.

6.1 Overview

Participation was used in this project in order to determine how documentation affects participation to open source projects. Activities reflect an effort to improve Apache Derby, through participating to it by contributing code and following the project's development.

The time used in the various phases of the participation is shown in table 6.1.

Phase	**Planning**	**Intervention**	**Evaluation**
Iteration 1	08.03	12.03	27.03
Iteration 2	28.03	28.03	19.04
Iteration 3	20.04	22.04	08.05

Table 6.1: Starting dates used for the various action research iterations

In addition to work on resolving issues, the researcher also followed mailing lists and IRC[1] channels for Apache Derby. Following the mailing lists imply a certain overview of project activity, as source code changes are automatically posted on a mailing list.

[1]Internet Relay Chat, a real-time text-based chat service

6.2 First iteration

The first phrase started with an understanding as stated in the problem description in the introduction. It was assumed that there are differences between what users and developers need from documentation of a product, and what kinds of documentation exists. This may be a consequence of that working on documentation can be viewed as an unrewarding task by the technicians who have learned the system from scratch.

The lack of sufficient documentation may hurt the adoption of open source, as products can be perceived as less mature, and as having less activity. Publicly available articles provide valuable exposure, and is one method to counteract this. In addition to documentation, it is important to have a strong community that can answer questions and react to changing demands.

As the development of Apache Derby is supported by major corporations, it has good premises for studying how open source meet market demands. Through its commercial ties it is expected to provide an active and resourceful community to do research in.

Interesting for this research phase it to firstly get acquaint with the project, find out which types of documentation is produced, and how it is maintained.

Actions that will be included in this phrase include:

1. Examine mailing lists. If this is a mature project, there will be activity from a significant base of developers, and also users. If many have use from this project, both users and Derby developers may be participants in discussions.

2. Examine project artifacts. This involves getting familiar with the structure of the source code and the main features. If there are good documentation for newcomer developers, it might indicate that either:

   (a) they have a healthy focus on getting new developers to participate, or
   (b) there are excess developers in the project that are willing to do these kinds of boring tasks.

3. Do work in the project by looking for tasks that may be fixable with outside contributors. The researcher may additionally find tasks to work on from own, practical experiences with using Derby. Examine the process of including and/or discussing features in the project.

4. Entering the project will not include an introduction of the researcher. Since persons are not studied, this will be regarded as a research-ethical choice.

5. Special care should be exercised with regard to how difficult it is to understand principles of the Derby machine, and why it is difficult or not difficult to understand. This will document the hypothesis on documentation's effect on newcomers.

The theory is that much end-user documentation is available, which is the result of the aggressive publicity Apache Derby has received, and also its use in different commercial products (IBM Cloudscape and Sun Java DB). Documenting it is therefore easy,

because it is tied to a market model. It is assumed that the project is affected by a lack of architectural documentation, as is common in open source. The code is viewed as sufficient to understand the system, and architecture descriptions are not formalized. This is similar to that the requirements engineering process is an informal process in open source. The effort of getting acquaint with project artifacts will probably include mostly reading code to understand the structure.

6.2.1 Intervention

The social code that the community follows is very evident when observing an open source project. Guidelines on how to communicate in mailing lists is one form of codification of this:

> *Respectful and considerate communities are one of the pillars of the Apache way. Please aim to provide constructive comments and do not denigrate others.*
> ...
> *Every contribution is worthwhile. Even if the ensuing discussion proves it to be off-beam, then it may jog ideas for other people.*

Complaints in the community addresses that user documentation in Apache Derby is insufficient for beginners. One user comments in the mailing list that the installation guide is "extremely buggy". As a response to this, a community member mentions a few efforts that are underway for improving user documentation, namely issues Derby-879[2] and Derby-913[3]

Issue Derby-909 Derby-909 is an issue that is marked as a *newcomer* entry, which encourages newcomers to the project to look at it. The problem is that SQL queries containing string literals have suboptimal performance. For the researcher, it was difficult to find a method to approach this issue, due to problems understanding how querying works, and a solution would require a rather creative approach to the optimization.

Issue Derby-1062 Issue 1062 concerns a built-in Derby function to compress a table. Compressing a database table becomes necessary if rows of varying sizes are inserted and deleted in the table, leaving unused space between table rows. The issue involves combining this functionality with a similar function, in order to reduce code. Still, the investigation of this problem serves mostly education, as it is difficult to understand.

The use of Derby- or database-specific terminology is a problem. The use of locks and row handling makes this an issue that requires some more experience with working with the system first.

[2]Derby-879: *"The Getting Started Guide is incomplete (section: Installing and working with Derby) and so not helpful to new users"*

[3]Derby-913: *"[WWD] Proposal to create and add Working With Derby, an activity-based tutorial document, to the Derby documentation set"*

Derby-836 Issue 836 was also worked on, which is related to the calculation of display widths of decimal columns. As this bug also requires a great deal of work with updating test cases, this is a lot of work. Finding the location of the error also proves to be difficult with little knowledge of the API. Working with this issue furthermore involves consulting ODBC, JDBC and SQL specifications on what the proper use of decimal columns are.

A fix is implemented. However, one problem that occurs is that a huge number of test cases will fail if the patch is applied. For this reason, the patch is not submitted at this point.

Derby-1136 The issue Derby-1136 is about data types, which also is addressed in previous work of the researcher. This makes the work easier. The problem is loss of precision in floating-point numbers due to internal conversion between 32- and 64-bit floating point numbers. This issue is easily resolved, and a patch is submitted to the bug tracking system, Jira.

One mailing list thread from January 2005 addresses the problems with architecture documentation, and demonstrates awareness of such issues:

> *The Cloudscape development model has always been light on design documents, the design was intended to be captured in the code, its JavaDoc and general comments in the code. Functional specifications were written and they are all reflected in the user documentation.*
> ...
> *My guess is that there is nothing that exists that would help people understand an overall view of the system, they may be some focused papers on certain "edge" areas.*

6.2.2 Evaluation

During the intervention, the biggest problem was understanding the overall structure of the project, and its classes and Java packages. Understanding the program flow requires a lot of study, especially in an architecture with many links between modules. Evaluating the quality of the code is also difficult when being in the role as an outsider, with little reference material to consult.

There are some birds-eye views on the Internet that presents some of the higher-level aspects of Apache Derby. These are, however scarce, helpful to understand some of the basic concepts of the architecture, but are insufficient on their own.

The documentation for users is through discussions on mailing lists and regular updates shown to be regularly updated. Efforts to document the architecture of Apache Derby are also being worked on, are generally supported by many community members, but insufficient to create any good architectural overviews. Why are there many small efforts to document isolated pieces of the software, and no effort to bring this into an unified description?

Basing on the activities of this phase, it appears that there is a lot of documentation available to developers in this open source community. Documentation for users and developers have different motivations, but architecture documentation may also be relevant for evaluators. The architecture descriptions are spread out over several sources, with different authors and focuses. This makes it difficult to assess which assistance the documentation offers for the developers, as the developers can not be considered one homogeneous group.

6.2.3 Reflection

Taking the role as a contributor in the project was difficult, mostly due to the complexities of the code. Documentation that could aid the participation was scarce. In an attempt to get better understanding, suggesting new features could be useful. More thorough investigation of documentation shows no surprising findings. Both JavaDoc, test cases (functional and regression tests), and bug tracking system specify the expected behavior of the system, but they are neither easily readable nor accessible.

Another cycle of action research will be done to try harder to contribute with code. The motivation will still be to take the role of a developer, while revealing information on the research questions. Also interesting to understand is what is needed to make the job of newcomers easier to join open source projects, and what support could make their jobs easier.

6.3 Second iteration

Documentation in the Derby project may be a second-rated activity, which is overshadowed by the desire to provide "documentation" through tests and JavaDoc. Much documentation, and even printed literature exist for Derby users. The need for architectural documentation, which is need to post-system development understanding, and for newcomers to open source projects, however, is not properly addressed according to the researcher's observations.

The complexity of a database system can be overwhelming for a newcomer to such a project. There seems to be a "critical mass" of developers working regularly on the project, which makes it interesting to join. Newcomers can here enhance programming skills, get feedback on their contributions, and participate in discussion around the direction of the product.

Theory started with documentation is made mostly for end-users, with little or no architectural descriptions being present. The current understanding is that documentation is JavaDoc plus tests, and smaller text-only descriptions of various concepts of the Derby engine spread among Wiki, web pages, and mailing list threads.

Theory for this iteration will be that the community will focus mostly on assistance on process issues, like i.e. how to create patches, and how to go by to get assistance. These are the community rules which control how everyone works. Through such a system,

getting familiar with the system is harder because the user need to look a lot of places to find relevant information. The distributed nature of the documentation may confuse developers that need to get an overview. Examining code is the only way to get a proper overview, as getting a thorough overview through mailing lists, bug system, sporadic articles and code comments is costly.

Actions for this phrase include primarily looking at reasons for the found data in the previous phase:

- Suggest functionality to add to Apache Derby, i.e. `SHOW TABLES` functionality. The purpose will be to continue learning more about the system, and also try to improve it.
- Get a proper list of all documentation resources, and to whom they are useful.
- Main focus: Why is documentation not aggregated in one source? Why is everything spread out?

6.3.1 Intervention

Issue Derby-1164 A tool that accompany Apache Derby is the *ij* command-line scripting tool. This tool allows users to connect to a database, run upgrades or other operations, and execute queries. One feature that is missing from this tool, which is suggested implemented by Derby-1164, is the ability to display lists of existing tables, views, procedures etc.

Implementing this involves first being able to modify command parsing. Commands and SQL statements are in Derby parsed using the JavaCC parser generator. This essentially generates a tree structure of an entered command, according to a grammar.

After some discussion and patch suggestions in the bug tracking system, a workable implementation is made. Most of the work on the patch is spent on finding the most efficient and maintainable problem solutions. In addition, some Derby participants also highlighted the need that the *ij* tool also has to be compatible with other database management systems, which involved additional work on the patch.

The complete interaction log for Derby-1164 is attached in appendix A.

Derby-722 Working on more maintenance issues. Derby-722 is about functionality that differs in various forms of result sets. Interesting here is to observe how test cases can relate to the project artifacts. The role of tests is to discover differences that occur during the evolution of the system, in order to bring small behavioral differences to the attention of the programmer. Tests are this way forced to be *alive*, and can not be ignored. This way, they are not prone to aging in the same way as any architectural description would.

Derby-1208 A simple fix for issue 1208 is submitted to the bug tracking system. This is an issue that is a minor inconvenience when testing. The nature of the bug report was that a Derby object got an invalid state in the case of SQL statements that emitted errors.

The biggest problem as a developer was again to locate the precise code block that the error occurred in. With practice, this task is easier, but still keeping track of all interdependencies proves to be a daunting task.

Documentation sources As an attempt to find which documentation sources are generally used in open source, a brief search of available resources is made. The following resources are found in the Apache Derby project, and found important to participation:

1. **Wiki**, `http://wiki.apache.org/db-derby/`. Contains i.e. discussion around prioritization of bugs, brief architecture overviews, links to articles and resources, proposals, and information for newcomers.
2. **Web pages**, `http://db.apache.org/derby/`. Contains i.e. information on the product, downloading, license, quick start, brief architecture overviews, and project news.
3. **Manuals**, `http://db.apache.org/derby/manuals/index.html`, API, getting started, reference manual, developer's guide, tuning guide, administration guide, tools/utilities guide, working with derby.
4. **JIRA bug tracking system**, `https://issues.apache.org/jira/secure/BrowseProject.jspa`, with discussion around bugs, and suggested solutions.
5. **SVN repository logs**, `http://svn.apache.org/viewcvs.cgi/db/`. Also include commit logs.
6. **Mailing lists**, `http://db.apache.org/derby/derby_mail.html`

The purposes of these are different. Web pages serve as a presentation tool to the outside. Wiki allows easy update of knowledge in the community. Bug tracking system carries information on the process for resolving bugs. SVN commit logs traces all the changes, the rationale for the changes, together with the bug tracking system, and so on. The mailing lists are in many ways the glue that keeps the community together, through discussion around project control and an informal requirements engineering process.

6.3.2 Evaluation and reflection

There seems to be basis for the hypothesis, after only working in the project, but not studying in-depth, that documentation seem to be good for the process issues, which is necessary to attract new developers. There is no lack of information, but the documentation that exists is rarely relevant for detailed work.

Action planned and performed in the project seems to take unreasonably long time, and much effort. This will not be sustainable if someone were to i.e. do the same work while employed in an organization. Using open source in a commercial context then is

difficult – more documentation should be available if they expect other organizations to start efforts to participate in the development. Anything else may lead to costly development for organizations that possess little knowledge of the product from before.

6.4 Third iteration

People, at least the researcher, don't know where to look for documentation. Impression that the production of documentation is very much ad-hoc, put into a framework, "because they must", and because it is a rule that documentation should be produced. For elitism cultures it is easy to ignore the rest of the community, especially when the community is tight-knit and there are few benefits from the openness.

The distributed nature of documentation makes it harder to understand the project at a first glance. Interested developers must follow the project for some time in order to understand where the development is heading. There is a very well-developed documentation structure for the users, but for developers participating it is important to utilize the time perspective to get familiarized.

For individuals are participating to open source, lack of open source experience is probably reducing performance. Theory is that it is a learning process, and when experience is built, the individual will get self-confidence and know what requirements are set to the code, including what requirements are set to participating in the planning of releases, and thereby acquiring the knowledge required to work efficiently.

The following actions are planned:

1. Suggest issues necessary for making a documentation framework in open source. The type of architecture documentation, the extent, and the purpose are all relevant to mention.

2. Reflect on if newbies need more than process support, in light of the participation that the researcher is doing himself.

3. Continue normal participation.

6.4.1 Intervention

Derby-1164 The iteration started with continuing the work on Derby-1164, which was an issue related to the introduction of `SHOW TABLES`-alike functionality. Some problems are discovered, i.e. that there are no uniform ways of returning the currently active schema in database management systems. This addresses the cross-platform compatibility of Derby, which appears to be a sore issue in the community. Also, returning the correct subset of data to the user, is choices that there are no good answers to.

Suggestions for documentation production The researcher's suggestion for how documentation, in particular architecture documentation can be viewed in an open source community:

- **Social dimension**: Following the development on the mailing list is an interesting activity, and probably something that most open source developers interested in contributing should do. What is provided here is necessary in order to understand the social dimension – why people participate, how they participate, normal politeness, etc.
- **Process dimension**: CVS/SVN[4] Commit logs and bugs and the likes are also a vital part of understanding what is going on, why it is happening, and the progress. These are open to the outside to provide evidence for activity and health of the community. Also, information on how to contribute, and the extent to which the community values contributions from outside, is here good.
- **Technical dimension**: Models, technical overviews, design rationales. These may be spread across multiple sources, such as bug tracking systems, mailing lists, etc. This is for the purpose of understandability not a good solution, but simplifies the maintenance and gives participants a streamlined approach to their code work.

Who should create the documentation? Documentation is an intermediary whose purpose is to support the primary goal, the software, and its development and maintenance should be integrated with the software's development [For02]. Existing practices are sufficient for the participants, but it might be advantageous for newcomers to have something that enables them to 'speed up' the learning process. This could be i.e. better overview, a developer manual that documents all aspects of the system, schematic overviews, dependency diagrams or the like.

Derby-1262 The derby-1262 issue was regarding incorrect behavior of LIKE statements. The optimization of SQL statements containing LIKE would result in that control characters be handled incorrectly. If strings had a tab character (ascii 0x09) on the place where the string matching occur i.e. `"asdf`**TAB**`jkl" LIKE "asdf%"`, then it fails.

The bug is approached by trying to find out which function actually fails. The error is located by following the call hierarchy in the code, a process that was time-consuming. The steps are the following, included to demonstrate a point:

1. LIKE-optimizations are evaluated in the LikeEscapeOperatorNode class.
2. Various uses of LIKE uses different kinds of optimizations. Some of them involves replacing the LIKE statement with the faster LessThan and GreaterThan operators.
3. When LessThan and GreaterThan operators are used, BinaryComparisonOperator nodes are used to indicate comparisons in the query tree.

[4]CVS and SVN are common version control systems.

4. When the statement compiler runs, the comparison operator is serialized into the compiled statement class. The constraints (LessThan/GreaterThan value) of the comparisons are written in order of selectivity, that is, constraints that selects the fewest rows are prioritized.
5. When the compiled code runs, a BulkTableScanResultSet is generated (or index scan result set, in case the table column is indexed).
6. The BulkTableScanResultSet passes on qualifiers (constraints) to scan controllers. Scan controllers return rows to the table scan class, and ignores rows that does not satisfy the constraints.
7. SQLChar.stringCompare seems to be manifesting the failure.

What is worth noting here is the complexity of a process to resolve bugs when prior knowledge is weak.

A fix for this is implemented, and submitted. In order to increase the chances of the patch being accepted, the number of changes is kept to a minimum, and test cases are updated and verified to run correctly.

6.4.2 Evaluation

Newcomers to advanced open source projects should have additional support, as this is a costly activity. Newcomers can have a lot of motivation to get into the project, may also be very talented, and easily understand how things work. A larger group of not-so-savvy people may need support in the form of mentoring, or maybe better for larger numbers, proper documentation on architecture.

There may be properties of open source projects that make them more or less susceptible to problems with attracting newcomers. It may be unwise to apply practices in open source without good reasons. Limiting the freedom in an open source project may work counter-productive, taking away the freedom to work in an agile manner. Documentation should for most purposes therefore be regarded as a supporting activity.

6.4.3 Reflection

There are many implementations of open source where documentation for developers will not have a good effect at all. Documentation may be something only beneficial in hi-tech environments, with little user interface, where work is done on a component that is valued for its reusability. Requiring that open source produce documentation in a certain way is not meaningful, as the agility of open source processes are vital for their success. Improvements to practice need to be the product of measurable improvements to the development process and post-deployment maintenance.

The decision to end the action research process in Apache Derby is made, since the findings are sufficient for discussing the research questions. Further activity in Apache Derby would probably not uncover more.

Chapter 7

Discussion

This chapter summarizes and analyzes the findings of this thesis, and will further explain what information is the foundation for answering the research questions. The findings will be analyzed using the following axes:

- How knowledge is shared in the open source community.
- What issues are perceived to exist in participation, as researcher observes from own participation, or discussed on mailing lists.
- What other open source literature says on the same topic.
- How an inexperienced developer's expectations conform to the documentation artifacts that already exists.

These axes are not in themselves the subject of the study, but are aspects that the research may address, and may contribute to understanding the research questions.

7.1 Success factors for open source documentation

Wohlin summarizes in [WRH+00] the basic requirements for being successful in software development. These include:

1. Understanding the software process and product.
2. Definition of process and product qualities.
3. Evaluation of successes and failures.
4. Information feedback for project control.
5. Learning from experience.
6. Packaging and reuse of relevant experience.

These goals address how the software development process learn from its experiences, and manage information regarding the product. In software engineering, a separate organizational entity called a *Experience Factory* can be used for capturing experience. Its motivation is to make previous experience available to software development projects [Din02].

Open source implements an ad-hoc knowledge factory. Knowledge is implicitly codified in mailing list discussions, and issue tracking. As an effect of the distributed nature of open source, there is vast amounts of information on which persons have worked on which issues. This enables expertise on a particular issue to be located through finding who has worked on a particular part of the program before. Dingsøyr in [Din02] compares this type of knowledge management to a "yellow pages" of who knows about what.

The activities in the researcher's Derby participation also faced similar problems, when working with extensions to the IJ tool. While working on the issue Derby-1164, some negative feedback was received as the extensions broke features of the IJ tool which they found useful. Little codified documentation, if any, would help in getting the correct understanding. The feedback was a social process in which knowledge is stored in the network of developers.

A premise for efficient open source development is outside contributions. Knowledge accumulated through the development process may be available to the community through other participants in the community, and some knowledge available codified in textual sources. Either of these are not necessarily easily accessible, and recovering design decisions becomes either a social or a creative effort.

Understanding success criteria for documentation in open source is difficult because it requires knowing open source's ability to retain its base of expertise, and knowledge of design decisions that have been taken. Open source retains this inside-knowledge in the community through that the original authors of the product often stick around. They are the senior members who review and give feedback to new developments, ensuring the architecture and community remains healthy.

It is evident that larger open source projects are doing well with their usage and their innovation, measured from the popularity of the system. Larger open source systems easily attract volunteers, while smaller and more specialized domains may struggle. This has i.e. been shown in [Mor05]. This may involve higher requirements for codifying knowledge in the form of architecture descriptions, and not only in the typical network of communications that traditionally follows open source.

7.2 Apache Derby

There are several sources of documentation in the Apache Derby community, of which few resemble design or architecture descriptions in the traditional sense. These are:

- **Mailing lists**. Mailing lists are one of the primary sources of knowledge sharing. The mailing lists are used by developers to communicate the activities being worked on. Questions regarding how various functions of Apache Derby works is common.

  When users ask for help on the mailing lists, they are often directed to relevant web resources that provide the information they are looking for. Following the mailing lists for a short period of time leads to a good impression of where information

is located. In addition to addressing user problems directly, the mailing lists are also excellent sources for growing the amount of information available to Internet search engines.

As a consequence of that discussions in mailing lists are the core of understanding what is happening, the developers need to pay attention to them for a longer period of time to understand what the current goals of the development is. This allows observers to see who is working on what, and participate in discussions on what next releases should contain.

- **Wiki**. The name Wiki means a web site that is easily modifiable by its users, and where each modification of the page is tracked. Information stored in Wikis are typically changing more often than other information. Apache Derby uses Wiki for:
  - Bug lists, such as high priority issues, or tracking the compatibility to JDBC and SQL.
  - Information on how to contribute, and how code contributions should be formatted.
  - Information on proposals, sub-projects, and brief overviews of design and architecture.

  This is knowledge that mostly support the development process, and to assist in understanding specific aspects of the code. The purpose of Wiki appears to be to transfer know-how inside the community, much like an Experience Factory, but completely optional. An example wiki page is shown in figure 7.1.

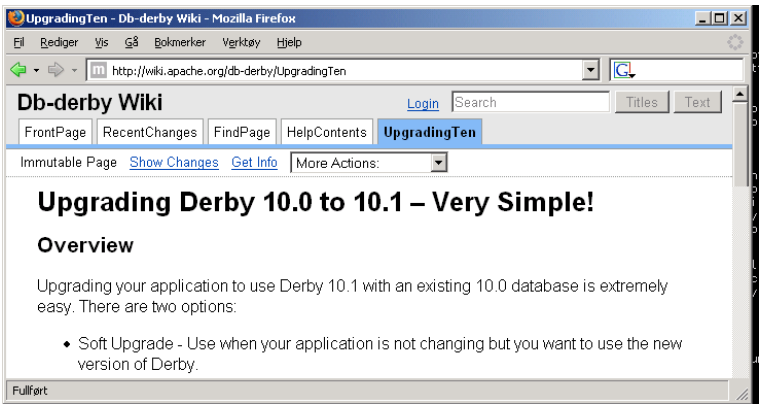

Figure 7.1: A wiki page for Apache Derby, editable by anyone.

- **Project artifacts**. The source code in Apache Derby constitute around 2.5MB of data, a number which also includes test cases and tools. JavaDoc in the code describes the interfaces between modules, and also some design- and architecture aspects. Through participation, this was found not to be sufficient to understand interdependencies between classes easily. An instance of this was Derby-1262, where

the researcher used considerable time to find execution paths through the program. The documentation describes design choices and explains classes, functions and algorithms, but fails to describe larger aspects.

Open source code is based on the principle that code can speak for itself. While intelligently written code has some ability to speak for itself, being a masterpiece doesn't necessarily make it easier to understand. Documentation inside the code is therefore easy to use if you know where to look, but is in itself insufficient.

- **User documentation**. Apache Derby provides considerable user documentation resources with information on installing and using the Apache Derby database system.
- **Test cases**. The expected behavior of the system is partly documented through its test cases. These are sets of programs or SQL statements, and their respective outputs. Before each check-in to the Apache Derby code, developers are required to run suites of tests that will verify that the changes doesn't break existing functionality.

  These are important in order to ensure that a compiled version of Apache Derby will be *binary compatible* with other programs. Binary compatibility means that a new version behaves similarly as an older version, with the exception of any additions to the interface.

  With regard to their relevance as documentation for developers, the test cases may be cursory for how the software should work. They address the historically evolving system, preventing changes that breaks existing functionality, while showing what different components of the system needs to support.
- **Bug tracking system**. JIRA, the bug tracking system used by Apache Derby, binds the source code to the implemented changes. This is another of the primary contact points with users. Entries in the bug system consist of a short description, problem description, and a participants' discussion of the issue.

7.3 Documenting software for newcomers

Especially through the work with issue Derby-1262, the researcher used a lot of extra time in trying to figure out architectures and execution paths through the program. This was a problem, there were no easy ways of getting an overview of the code except for relying on creativity, or asking experienced developers.

The lesson learned after the Apache Derby participation is that it requires a lot of time and motivation to learn the various aspects of a system. The workload required to learn a system may be a deterrent for contributors, especially those without well-defined incentives to contribute. Learning the open source process is easy enough even to learn through observation, but understanding deep function call hierarchies has some

problems. An hour spent on describing these can hypothetically save an hour for a lot of other developers.

Les Gasser et al. described the knowledge development/transformation process in open source as having *"extreme multiplicity of viewpoints, representations, experiences, and usages"* [GRSP03]. This multiplicity has been observed also in this study. Is this positive for a newcomer developer in open source? A likely effect is that parts of the software that evolves rapidly will through this be well documented or explained in bug tracking system or i.e. Wiki, while stable parts of the code will have no documentation usable for later maintenance.

From the study, Apache Derby and possibly other open source projects have a different idea of the requirements to document software. Software architecture literature [CBB+02] cites its decisive role in software development: "[documentation] is the conceptual glue that holds every phase of the project together for its stakeholders". The notion of open source being weak on producing documentation observed in this work and by others [PC04, Sta99, Ols06] is something that need to be recognized. However, there are counter-examples in open source, i.e. the Mozilla project described in [MFH02].

While code reading and tool support for retrieving architecture offers some help, the mental knife sharpening and rigor introduced to the development process through architectural descriptions and traceable requirements is a possibility that should get more focus.

7.4 Comparing findings to other literature

Development of software with open source can be more productive, give higher quality, and cost less than counterparts in commercial software development [GRSP03]. This suggests that while open source is light in its process, it still delivers successful software. However, there are good arguments for bringing open source closer to software engineering practices, which involves a disciplined approach. Having a rational software development process involves that architectural descriptions and planning is done in advance of implementation, that the system is open to easier reviews and transfer of people and ideas, and that progress can be more easily tracked [PC86].

The complexity of software-intensive systems are dependent on the number of programmers working on it [Web06]. The product is not necessarily the the sum of all developers' work, as human communication diminishes when traveling between large numbers of people (ibid). Coping with this problem requires effective communication where perspectives from inside and outside both are regarded. Traditional open source development manages communication well, but there is potential for improvement through educating participants on the need for storing knowledge and documenting architecture.

Chapter 8

Conclusions

In this work we have studied how open source relates to production and maintenance of documentation. Understanding the properties of the products being developed, as well as the process, are success factors for any software engineering project. For open source, understanding the product requires a long track record of following the product's development.

The participation to Apache Derby, in addition to experience in participating to open source, resulted in good information on how the system is documented. Relations between the different artifacts that are managed in the project were found, like Wiki, issue tracking and test cases. All serve the purpose of tracking changes, and sharing knowledge among the participants. The artifacts were also found to be categorizable into social, process, and technical dimensions.

The core of this discussion, the lack of architectural documentation, is an issue that concerns large open source projects. While Apache Derby is strong on providing information for its users, the lack of a documented architecture leads to higher learning costs for newcomers. This does not constitute a problem unless central actors leave the project. The product's maintainability and quality would be jeopardized if code reviewers have little knowledge of the architecture and design considerations.

Open source managers should be aware that running a process light on documented architecture may have negative consequences in projects with marginal diffusion. The luxury of agility and document-less development may be better suited to high-status, high-profile open source projects, where there are greater social incentives for experienced developers to stay. Furthermore, any documentation efforts in open source should be integrated in the development process itself, in a manner similar to the integration of the issue tracking system or test cases in Apache Derby.

We have discussed the use of documentation in context of an action research study. Knowledge and self-consciousness about open source development processes is based on maturation, and is a prerequisite for experimenting with open source. These findings are based on hands-on experience, but are only one person's opinions in context of one open source project.

Current practice in open source is evidently sufficient for creating creating innovative,

new software. Experimenting with the production of birds-eye views on architecture, or design documents, would be interesting. This could ease contribution barriers for newcomers, and allow easier understanding of the product qualities that the community values. Apache Derby has many efforts to document its intricate architecture, but they are not aggregated.

It would be unwise to prescribe practice for open source communities. However, the awareness of how knowledge is shared would remove some of the fuzziness of open source, and bring it closer to formal software development processes.

Further work

Using action research for learning and researching open source has been a successful endeavor. In an educational context, it gives the student opportunity to reflect on software engineering, work on technical challenges, and also contribute to knowledge.

The findings in this work are based a qualitative study of the Apache Derby project. There is a need to investigate how other open source projects manage knowledge and documentation in general. Making quantitative studies on this would increase confidence in the results.

Based on experiences from this work, or from ideas from using the research approach in this study, here are some suggestions for future research:

1. This work suggests that some open source projects have a need for a documented software architecture. For existing projects, recovering the architecture may involve understanding both how the development is performed (a development view), and recovering a logical structure of the software. It may be interesting to devise methods for architecture recovery in the context of open source.

2. Suggested in this work is that documentation should be integrated with the open source development process in cases where documentation is regarded as important. How can this best be done, and what cost will such an integration have?

3. What risk factors are there for companies wanting to use open source as a development model? Adopting open source processes involves some risk, notably the risk of giving away code with no benefits for the owner. Which factors need to be in place for a company wanting to reach a particular goal through maintaining an open source project? Is open source predictable enough to be called an engineering tradition?

4. Larger open source communities can be divided into virtual work groups that work on isolated parts of the code. For instance, the Mozilla project consist of teams managing various technologies such as RDF, XPFE, XUL, JavaScript, and HTML rendering. It also manages 50+ localization teams. How is knowledge shared among such sub-groups, and how do they ensure that the product is coherent? How does the development process encourage cross-team communication?

5. A participatory study in open source could also address how quality attributes are manifested in open source projects, as they otherwise are presented in software architecture literature. Are software designed with modifiabilty and reliability in mind, or is this a result of an imminent need for some party in the community?

Bibliography

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[BF04] Wolf-Gideon Bleek and Matthias Finck. Migrating a development project to open source software development. In Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors, *Collaboration, Conflict and Control – Proceedings of the 4th Workshop on Open Source Software Engineering*, pages 9–13. International Conference on Software Engineering, May 2004.

[BH99] Ivan T. Bowman and Richard C. Holt. Reconstructing Ownership Architectures To Help Understand Software Systems. In *Seventh International Workshop on Program Comprehension*, pages 28–37, 1999.

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 555–563. International Conference on Software Engineering, May 1999.

[BP01] Erik Berglund and Michael Priestley. Open-source documentation: in search of user-driven, just-in-time writing. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 132–141. ACM Special Interest Group for Design of Communications, May 2001.

[BR02] Andrea Bonaccorsi and Christina Rossi. Why open source software can succeed. *Research Policy*, 32(7):1243–1258, July 2002.

[BZ99] Izak Benbasat and Robert W. Zmud. Empirical Research in Information Systems: The Practice of Relevance. *MIS Quarterly*, 23(1):3–16, March 1999.

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures*. Addison-Wesley, 2002.

[Deb04] Daniel J. Debrunner. Internals of Derby. `http://www.softwaresummit.com/2004/speakers/DebrunnerDerbyInternals.pdf`, 2004.

[Dic93] Bob Dick. You want to do an action research thesis? http://www.scu.edu.au/schools/gcm/ar/art/arthesis.html, 1993.

[Din02] Torgeir Dingsøyr. *Knowledge Management in Medium-Sized Software Consulting Companies*. PhD thesis, Norges Teknisk-Naturvitenskapelige Universitet, 2002.

[DMK04] Robert M Davison, Maris G Martinsons, and Ned Kock. Principles of Canonical Action Research. *Information Systems Journal*, 14(1):65–86, Jan 2004.

[DSV03] Jianjun Deng, Tilman Seifert, and Sascha Vogel. Towards a Product Model of Open Source Software in a Commercial Environment. In Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors, *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 31–37. International Conference on Software Engineering, May 2003.

[For02] Andrew Forward. Software Documentation – Building and Maintaining Artefacts of Communication. Master's thesis, University of Ottawa, Ottawa, Ontario, K1N 6N5, Canada, 2002.

[Fou99] The Apache Software Foundation. How the ASF works. http://www.apache.org/foundation/how-it-works.html, 1999.

[GRSP03] Les Gasser, Gabriel Ripoche, Walt Scacchi, and Bryan Penne. Understanding Continuous Design in F/OSS Projects. *16th. Intern. Conf. Software & Systems Engineering and their Applications*, Dec 2003.

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 2000 International Conference on Software Maintenance (ICSM '00)*, pages 131–142, San Jose, California, USA, Oct 2000. IEEE Computer Society.

[Hen05] Joachim Henkel. Patterns of Free Revealing – Balancing Code Sharing and Protection in Commercial Open Source Development. *EURAM Conference 2005*, Aug 2005.

[HNH03] Guido Hertel, Sven Niedner, and Stefanie Herrmann. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy*, 32:1159–1177, 2003.

[IEE90] IEEE. *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*, 1990. IEEE Std 610.12–1990.

[IEE00] IEEE. *IEEE Std 1471 Recommended Practice for Architectural Description*, 2000. IEEE Std 1471–2000.

[Ini05] Open Source Initiative. Open Source Definition. `http://www.opensource.org/docs/definition.php`, 2005.

[Jor89] Danny L. Jorgensen. *Participant Observation – A Methodology for Human Studies*. Sage Publications Ltd., 1 Oliver's Yard, 55 City Road, London, EC1Y 1SP, UK, April 1989.

[Koc04] Stefan Koch. Agile Principles and Open Source Software Development: A Theoretical and Empirical Discussion. *Lecture Notes in Computer Science*, 3092:85–93, Jan 2004.

[Kru95] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, Nov 1995.

[LT00] Josh Lerner and Jean Tirole. The Simple Economics of Open Source. *The Journal of Industrial Economics*, 2:197–234, Dec 2000.

[Maa04] Wolfgang Maass. Inside an Open Source Software Community: Empirical Analysis on Individual and Group Level. In Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors, *Collaboration, Conflict and Control – Proceedings of the 4th Workshop on Open Source Software Engineering*, pages 65–69. International Conference on Software Engineering, May 2004.

[MD04] T.R. Madanmohan and Rahul De. Open Source Reuse in Commercial Firms. *IEEE Software*, 21(6):62–69, Nov 2004.

[MFH02] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[Mic05] Sun Microsystems. JSR 221: JDBC 4.0 Specification, public draft. `http://java.sun.com/products/jdbc/`, 2005.

[Mor05] Håvard Mork. Leadership in Hybrid Commercial-Open Source Software Development. Directed study, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, Dec 2005.

[Moz06] Mozilla.org. About Mozilla. http://www.mozilla.org/about/, 1998-2006.

[Ols06] Michael Olson. Dual Licensing. In Chris DiBona, Danese Cooper, and Mark Stone, editors, *Open Sources 2.0: The continuing evolution*, pages 71–90. O'Reilly, 2006.

[Østerlie06] Thomas Østerlie. Producing and Interpreting Debug Texts. In *Proceedings of the 2006 Open Source Software Conference*, 2006.

[Pan96] Naresh R. Pandit. The Creation of Theory: A Recent Application of the Grounded Theory Method. *The Qualitative Report*, 2(4), Dec 1996.

[PC86] David L. Parnas and Paul C. Clements. A Rational Design Process: How and Why to Fake it. *IEEE Transactions on Software Engineering*, 12:251–257, Feb 1986.

[PC04] Vidyasagar Potdar and Elizabeth Chang. Open Source and Closed Source Software Development Methodologies. *Collaboration, Conflict and Control – Proceedings of the 4th Workshop on Open Source Software Engineering*, pages 105–109, May 2004.

[Ras00] Chris Rasch. A Brief History of Free/Open Source Software Movement. `http://www.openknowledge.org/writing/open-source/scb/brief-open-source-history.html`, 2000.

[Ray00] Eric Steven Raymond. The Cathedral and the Bazaar. `http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`, 2000. Revision 1.57, 2000.

[Ray01] Eric Steven Raymond. How To Become a Hacker. `http://www.catb.org/~esr/faqs/hacker-howto.html`, 2001. Revision 1.34, 2006.

[Sca04] Walt Scacchi. Free and Open Source Development Pratices in the Game Community. *IEEE Software*, 21(1):59–66, Feb 2004.

[Sen04] M. Senyard, A.; Michlmayr. How to Have a Successful Free Software Project. In *Software Engineering Conference*, pages 84–91, Dec 2004.

[SSR02] Srinarayan Sharma, Vijayan Sugumaran, and Balaji Rajagopalan. A framework for creating hybrid-open source software communities. *Information Systems Journal*, 12(1):7–25, 2002.

[Sta99] Richard M. Stallman. The GNU Operating System and the Free Software Movement. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly, 1999.

[STT01] Dennis Smith, Bill Thomas, and Scott Tilley. Documentation for Software Engineers: What is Needed to Aid System Understanding? *The 19th Annual International Conference on Systems Documentation (SIGDOC 2001)*, 2001.

[TT03] Sigurd Tjøstheim and Morten Tokle. Acceptance of new developers in oss projects. Master's thesis, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, June 2003.

[VV04] Kris Ven and Jan Verelst. Control Objectives in Open Source Projects. In Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors, *Collaboration, Conflict and Control – Proceedings of the 4th Workshop on Open Source Software Engineering*, pages 100–104. International Conference on Software Engineering, May 2004.

[Wal06] Stephen R. Walli. Under the Hood: Open Source and Open Standards Business Models in Context. In Chris DiBona, Danese Cooper, and Mark Stone, editors, *Open Sources 2.0: The continuing evolution*, pages 121–135. O'Reilly, 2006.

[Web06] Steven Weber. Patterns of Governance in Open Source. In Chris DiBona, Danese Cooper, and Mark Stone, editors, *Open Sources 2.0: The continuing evolution*, pages 361–372. O'Reilly, 2006.

[Woo05] Woods, Dan and Guliani, Gautam. *Open source for the enterprise*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, Aug 2005.

[WRH+00] Claes Wohlin, Per Runeson, Martin Hőst, Magnus C. Ohlsson, Bjőrn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, 2000.

[Zaw99] Jamie Zawinski. Resignation and Postmortem. `http://www.jwz.org/gruntle/nomo.html`, 1999.

[ZSB05] Paul C. Zikopoulos, Dan Scott, and George Baklarz. *Apache Derby – Off to the races*. Pearson Education, Inc., 2005.

Appendix A

Issue log for Derby-1164

Names are anonymized due to only the process is of significance, and not the persons.

Description
New users migrating from mysql are familiar with commands 'show tables' and 'describe' to respectively display all permanent tables, and show fields in a given table. These are not standard sql, but I suggest to implement them only in the IJ tool for user-friendliness.

As suggested in db-dev, using DatabaseMetaData should provide the necessary query strings.

Comment by Håvard Mork [30/Mar/06 12:41 PM]

Attached suggested patch. I am thinking outputting only a minimal information may be sufficient, and very convenient in order to reuse existing code. Alternatively, for 'describe', separate SQL queries could be made (with ijStatementResult) to output data in a table format.

Feedback is appreciated.

Comment by C1 [30/Mar/06 05:07 PM]

Hi Håvard,

Thanks for this useful patch. The parser productions look good to me. However, I'm hesitant about modelling the result as a vector. I think it would be better to return results which look more like tables rather than coalescing all the information into a single column. I think you will end up with something more powerful if you write an implementation of ijResult which wraps a ResultSet. This will let you rapidly display all sorts of DatabaseMetaData calls.

Comment by C2 [31/Mar/06 01:41 AM]

Thanks for this useful patch!

Did a quick review, with following comments:

1) Should IJ help be modified to show new commands?
2) If you have the itch, more could be added... to display
   procedures/functions, indexes, views, synonyms...
3) Should a test be added to function tests? It may be little more involved
   to add test than to code this!

Comment by C3 [31/Mar/06 09:44 AM]

Nice start. I think some more work is needed to make this really useful:

1. Display length for character types, precision for numerics
2. Display information on primary keys and nullability
3. Not require schema name, and look for tables in current schema in that
   case. I think many of those who will benefit the most from this command,
   may not necessarily know much about schemas. They just use the default
   schema.
4. Would be nice to have a 'show schemas' command
5. Would ne nice to be able to list tables within a given schema.

Comment by Håvard Mork [06/Apr/06 05:51 PM]

Thank you for all your feedback. I've thought more about this, but I've
been unable to find a good way to using the returned resultsets from
DatabaseMetaData directly, while still getting a nice layout.

In the attached patch, I have made local versions of the DatabaseMetaData
query strings. Where possible, I've tried to cast data types so rows will
fit on 80-char display widths. As for the querying parts, I'm unsure
whether this is the best approach.

Comment by C4 [06/Apr/06 06:24 PM]

Copying the meta data queries into ij ties ij directly to a specific
version of the database engine. I don't believe this is a good direction.

ij is intended as a fairly neutral JDBC client that works against any
database engine. While it may be ok to tailor ij for Derby, having to
need a specific version of ij to talk to a specific version of Derby
will cause problems.

It would be interesting for you to provide a patch that uses the DatabaseMetData methods directly and displays the resultant ResultSets using the standard ij code. The we could think about the "nice layout", what's nice to you maybe horrible to someone else.

Comment by Håvard Mork [07/Apr/06 11:52 PM]

Thanks for the review. I agree with what was pointed out regarding the intended use of ij.

One thing that puzzled me was the 'getTableTypes' and 'getTable' entries in metadata.properties. They both mention tables, system tables, and views, but they have no entries for synonyms. I don't understand issues regarding version compatibility of metadata.properties well, so please say if there are any problems.

Comment by Andrew McIntyre [20/Apr/06 05:07 AM]

Hi Håvard,

This patch is looking pretty good. A few comments:

- the ij help command should be updated to include a description of the DESCRIBE command.

- better error handling for bad syntax and validating proper input of table names would be nice, but not a necessity. For example, try entering in bad schema or table names: 'show tables in blah;' or 'describe blah;' where blah doesn't exist. Again, not a necessity. It might be more trouble than it is worth.

- a 'show indexes' command (corresponding to DatabaseMetadata.getIndexInfo) for a table would be nice, but not a necessity for the patch to be committed. That could be added later, same for showing primary/foreign keys on a table.

- as for displaying the results nicely, perhaps you could alter ijResultSetResult so that it takes two arrays, one which specifies which columns from the metadata query to present, and the second which contains the width at which to print the corresponding column. These arrays could then be set appropriately from the show* methods in ij.jj as needed. Then you could alter JDBCDisplayUtil so that when it takes an ijResultSetResult to display, it gets the arrays it should display from the columns array of

the ijResultSetResult (or all columns if the column array is null), and only displays them at the width specified in the second array (or the default width if no width is specified in the second array). I realize that this could end up being a fair piece of work, but it might be worth the effort. Does anyone else have any good ideas (meaning simpler) for presenting the results nicely?

C2, the change to the metadata query looks ok. Is there a reason why synonyms where not added to the results returned by getTableTypes() earlier?

Comment by Håvard Mork [26/Apr/06 09:08 PM]

Your suggestions are very helpful. I am attaching a new patch (1164_4.diff) with some changes:
* Modified JDBCDisplayUtil to display a selection of columns, with column widths set by ij.jj
* Added 'show indexes'
* Verifying that tables exist

There are still some problems:
* There are no standard way of getting current schema. Now using 'values current schema'. For non-Derby dbms-es, the failure to return current schema will lead to that i.e. DESCRIBE will return matching table names in all schemas (with a SCHEMA column, though). Any smarter way to solve this?
* What information to print and column size is a subjective matter. It is difficult to choose between one-line-fits-all, and ensuring that column values are output fully.

As for viewing fk/pk in tables, I've chosen not to implement it now, but that could be implemented later.

The patch passes derbytools and derbylang.

Appendix B

Article: Studying Open Source with Action Research

The following article was written in conjunction with the candidate's autumn project and master's thesis. The chapters 2.1, 2.2 and 3 are written by the candidate.

Studying Open Source Software with Action Research

Letizia Jaccheri
Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
letizia@idi.ntnu.no

Håvard Mork
Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
havard.mork@gmail.com

ABSTRACT

Open source projects can be regarded as interesting sources for software engineering education and research. By participating in open source projects, students can improve their programming and design capabilities. Master's students can also contribute to increased knowledge concerning research questions by reflecting on their own participation with an established research method and plan. This work reports on a study in the context of the Netbeans open source project, which serves as a successful example of using students for educational and research purposes. The research method used is action research.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Management
; K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms

Experimentation

Keywords

Open source software

1. INTRODUCTION

Open source software development poses serious challenges not only to the commercial software industry but also to academic institutions that have the mission to educate software engineers. Some of our students love to participate in open source projects and some of them have been participating in open source projects for years, motivated by their passion for programming. How do we, as software engineering teachers and researchers, tackle this challenge so that we provide a sound and motivating milieu for software engineering education?

Open source software projects have been exploited for successful empirical software engineering research [15]. At the same time, the importance of open source software as a trend which should shape our education program is well accepted [20].

There is a tradition for combining empirical software engineering research and education [4]. In this work we aim to exploit open source software for education and empirical purposes.

This article discusses the use of action research [6] in the context of empirical open source software research. We use action research for organizing open source education and empirical research. Assignments are designed for fifth year students, who act as both researchers and developers in open source projects. The assignment is to participate in one open source project. The main constraint and source of feedback, in addition to teacher supervision, is the interaction with one existing open source project. The student influences what kind of technology that is worked with. The goals of the project include defining relevant research questions. The candidate will have to study existing literature on open source development, select an open source project by defined characteristics, and participate actively in this project. Students can choose this assignment as part of their ninth semester project (500 hours, which counts half of the teaching load that semester) or tenth semester master's thesis (1000 hour project, which is the total teaching load for the semester).

Students who are admitted to these projects must attend two supporting courses (with exam). The first course provides a bulk of open source literature ([2, 3, 9, 10, 15, 17, 19, 22, 23, 24]), and they have access to the reports of the students who have previously worked on similar projects. The second course is an introduction to empirical software engineering [5].

In this paper, we will present a case that will provide the choice of the project, the research questions and the answers we found to them. More important, we will summarize some lessons we learnt from action research and open source software. We discuss how the choice of the research questions, the research methods, the literature, and the choice of the open source project are dependent on each other.

The structure of this paper is as follows: Section 2 provides the foundations of this work in three fields which are open source software, action research, and the intersection between software engineering education and empirical software engineering. Section 3 presents our case and Section 4 provides discussion and our conclusions.

The 5th International Symposium on Empirical Software Engineering ISESE 2006 Conference September 21st-22nd, 2006 Rio de Janeiro

2. BACKGROUND

2.1 Open Source Software

Open Source Software (OSS) is a concept that does not represent one particular software engineering method, but can be more accurately described as a set of principles, or a philosophy. Open Source (OS) means software that is developed according to a license model that conforms to the open source definition[1].

Software developed as OSS differs from commercial software in both how the product undergoes changes, and the economic incentives for the development. Commercial software is developed with private benefits in mind. The product is then the result of a requirements engineering process, which is aimed at creating value for potential customers. On the other hand, open source operates with the goal that anyone can alter and redistribute the software with no economic compensation to copyright holders.

The success of open source can largely be attributed to the "scratch your own itch" ethos. Developers who need to solve a particular problem may turn to open source in order to see if a similar problem has previously been addressed. If not, and they think the problem is shared with others, a new open source effort may be the solution, so the developer can benefit from contributions from people working with the same issues. Open source can thus be a solution for the need for a common toolbox.

2.1.1 History

The history of open source begins with the hacker culture of the 1970s and 1980s where making source code freely available was common in academic communities. In 1985, the Free Software Foundation (FSF) was established by Richard Stallman as a reaction to the "closed source" culture of commercial software vendors. The intentions were to bring freedom to copy and modify software back to the average user. Finally, in 1998, the term "Open Source" was introduced as a concept that disassociated itself with the anti-commercial ties of "free software". The difference between these two terms are mostly ideological.

2.1.2 Volunteer culture

Developers who participate in open source projects consist of a very diverse group of people, both professionals and hobbyists. Their mutual interest in the product of a specific OS project is what connects them. For many developers, OSS represents a proving ground where they can have varying learning goals, enhance social relationships, and gain privileged access to a community [22].

On a group level, trust and collaboration are basic principles. Members collaborate on the basis of mutual expectation, which either leads to trust or suspicion [14].

The potential for commercial use of the open source phenomenon is also apparent. OSS represents an easily accessible platform for innovation [7]. With open source, small firms can reduce the cost of innovation, with the possible disadvantage of sharing their own ideas with competitors. It also enables service-oriented architectures to spread on the web, where content of different nature can be integrated, like human-readable text, pictures, and interactive components. Perhaps the greatest economic contribution of open source is the new services it enables [17].

OSS is traditionally developed by communities of developers. These communities consist of individuals with a common goal, but with varying reasons for participating. The participants are typically not assigned to tasks, but focus on aspects of the product in which they have interest or expertise. The fact that the developers generally also are the users of the software implies that the requirements elicitation process in open source will be the result of the personal agendas of the contributors [19].

2.1.3 Development practices

Development in open source is characterized by being an evolutionary process, which differs considerably from commercial processes. Typical commercial models involve lifecycle models where phases in the development are sequenced. A different approach is taken in open source, where development consists of concurrent processes that all work towards a goal [19].

The highly distributed nature of OSS also has implications for how OSS works. Developers may live in different time zones, may have day jobs to attend to, while other developers may be paid by an employer to do OSS work full-time. With the various time slots generally available to individuals, asynchronous communication is necessary. Mailing lists and newsgroups are commonly used to coordinate communication.

Deng et al. points out that project management in open source is reduced to a minimal set of "technical" activities, that revolves around which code to include and not include in the main product [7]. The decision power in OSS generally arises from the merits and previous contributions of the members.

2.1.4 Licenses

Unlike commercial, non-free software, open source gains much of its momentum in that it is available for no cost to adopters. Some firms have managed to create commercial models around OSS, by offering complementary products and services. The role of software licenses is here to protect open source from opportunistic behaviour. Licenses protects the ownership of the code, limiting how the product is distributed and ensuring that it stays free. The licenses are furthermore required by the open source trademark to abide by the principles of the open source definition.

The GNU Public License[2] is one license that conforms to the open source definition. This license commands that the source code needs to accompany all versions of the software, that anyone can make changes to the code, and that the altered version must carry the same license agreement.

2.1.5 Leadership

The basic principle of open source is that anyone can download, use, and modify software without having to pay. This philosophy works because the product will be improved through collaboration and adapted to other needs [3]. Deciding which improvements and changes that should be implemented into the main version is up to members of the community to decide.

Authority in open source communities is based on member's merits. Developers that have contributed a lot to a

[1]The Open Source Definition is available at http://www.OpenSource.org/. The term "Open Source" is a trademark of the Open Source Initiative.

[2]http://www.gnu.org/copyleft/gpl.html

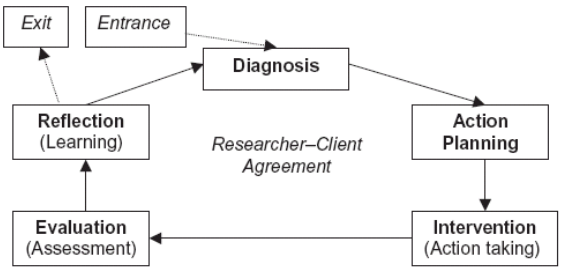

Figure 1: The cyclical process model (CPM) of CAR.

project will have significantly more influence than users that just participate on mailing lists. Only a limited number of persons will typically have access to make changes in the source code, in order for the quality of any changes to the software to be controllable.

2.2 Action Research

The empirical study that is reported on in this work is based on the action research (AR) method. AR originates from social sciences, and is used for learning from experience by intervening in a system. One orientation of AR is Canonical Action Research (CAR), proposed by Davison et al. [6].

CAR, according to Davison's description, is an iterative process which consists of two main components: carefully planned and executed cycles of activities, and a continuous process of problem diagnosis. It has a dual intention of improving practice in an organization through a change process, while contributing to knowledge about the object of the study.

A principle of applying CAR is having a theory in advance for doing CAR research [6]. This is helpful to position the work in a cumulative research tradition, narrow the scope of the research, and ensure that the work has scholarly as well as practical interest. While CAR may be applied without theory ("action learning"), the amount of collected data may be reduced if a theoretical framework is already established.

The various phases of a Canonical Action Research iteration are shown in Figure 2.2. The phases are:

1. Diagnosis: This phase involves the researcher diagnosing the organizational situation and reflecting on its causes. The objectives of the CAR project will control what is studied here, along with experiences from previous CAR iterations. Goals of the diagnosis phase include determining causes of a problem, and study the environment in order to allow a thorough action planning phase.

2. Planning: The planning phase should generate a course of action for examining the subject and collecting data. The actions that are planned should be related to the objectives of the AR project, as well as the current understanding of the problem being examined. The motivation for the actions will be to better understand the problem at hand.

3. Intervention: The purpose of the intervention phase is to apply change, and observe the outcome. Together with theories and expected results, this represents the main data source for later evaluation stages. Intervening in an organization requires that a plan for collecting data is present. Data collection techniques should be applied before, during, and after the intervention, in order to ensure a large volume of data is available for later analysis.

4. Evaluation: Results from the intervention phase should be analysed in the context of the current understanding of the problem and the goals of the research.

5. Reflection: Reflecting on the results of an iteration can determine whether additional iterations are necessary, or the lessons learned can be used to further refine research questions. If the goals of the project have been accomplished, then it could be decided to terminate further investigation.

2.3 Software engineering education and empirical software engineering

The software engineering education community values industry interaction and dialogue about what and how we should teach to our students [13]. At NTNU[3], IDI[4], we have 30 years experience of working in cooperation with the Norwegian software industry [1, 11] in the context of student projects.

Open source software development is relevant to industry as many companies are participating in open source projects and using open source software. The education community accepts this trend and recognizes open source software as a source of inspiration and influence for software engineering education [20].

In [4] we propose a framework in which empirical software engineering and software engineering education can co-exist by defining four roles that have some interest for student projects. These roles are: teachers, students, researchers and industry. A similar perspective about synergic aspects of empirical software engineering and software education is discussed in [18].

[3]The Norwegian University of Science and Technology
[4]The Department of Computer and Information Science

3. THE CASE

The directed study reported on in this work, which is available in [16], was based on an assignment that did not constrain the student with regard to what research questions would be posed. Only that Action Research should be used, and participation to an open source project was mandatory. This study was carried out by one student.

Previously at NTNU, there has been one master's thesis following the same kind of assignment [21]. This master's thesis was carried out by two students working together, and served as an example for participatory research for the student reported on in this case.

3.1 Diagnosis

The project aimed to participate in and contribute to open source software development in order to better understand how firms use open source for their own software development. The goal was to determine the effects of using formal techniques in open source projects, like explicit planning, ownership, inspection and testing in open source projects, as they occur in commercially controlled open source projects. Through the study, it was intended to see if the commercial use of OSS leads to a more manageable process.

The research goal and questions were created from a literature survey that was done in the first two months of the project.

3.1.1 Research questions

Two research questions were formed:

Q1: *Are developers who are not directly hired by the controlling organization able to affect the decision processes?*
The rationale for this research question was to uncover how commercially operated OSS projects view volunteer developers. In case of any confusion of roles, especially with regard to paid vs. non-paid developers, the current OSS development process may not be well implemented.

Q2: *How much of the decision process is open to the whole community, and to what extent are decisions taken inside the organization that is controlling the open source project?*
For OSS projects where decisions are not multilateral, participants may feel there are conflicts in the community, as described in [12].

3.1.2 Project selection

As the number of potential open source projects to choose from was large, a project selection phase was initiated in order to find a project that suited the study well. Project selection and research questions are related, as the studied artifact must be suitable for the research goal. Open source communities have widely different differing characteristics; they have varying size, different goals, and can accomplish higher levels of maturity. As the project aimed to investigate commercial ties in OSS, the selection process aimed to find a project in which this connection was prevalent.

A list of OSS projects where the project maintainers were known to be commercial firms was generated from Internet searches. These were Jetty, JBoss, PHP, Mozilla, MySQL, JINI, SugarCRM, and the one later selected, the Netbeans IDE. The projects were subsequently evaluated in context of a set of selection criteria:

- Should consist of 10-50 active developers, observed from public mailing lists or bug tracking system.
- Community allows entrance in a supporting role.
- Formal techniques (project planning, etc.) are used in the OSS development.
- Implementation is done in either Java or C++, to which the student is acquainted.
- Available public mailing lists, chat, and bug tracking.
- Software has general usefulness for student.

From the student's evaluations, Netbeans, Mozilla, and JINI accomplished similar scores. Mozilla was not selected because the student already was active in this community. Prior experience in an OSS community, however, should not pose any problems with the possible exception of reduced personal learning. Netbeans was subjectively evaluated to be more interesting than JINI, and therefore Netbeans was selected to be the subject of the study.

3.2 Planning

During the initial planning phase of the project, a data collection strategy was developed. With qualitative data analysis, the goal was to capture as much interaction with other participants as possible, thoughts and opinions during the project. The following elements were emphasized in the plan:

1. Which people that participated in discussions in the community.
2. The process which is used for accepting or discussing contributions, and how decisions for accepting code changes are made.
3. Communication between developers regarding changes will be useful for later analysis, to see how decision-making processes work.
4. Information about how source code contributions fit into schedules and personnel allocation.

The initial theory for the project was based on an assumption that the commercial use of open source will include practices from both commercial and open source development processes.

3.3 Intervention

The study was executed with two iterations of Action Research. The student started with little knowledge of the decision processes in the Netbeans project. A meritocratic[5] leadership was assumed to exist in addition to the maintainer organization's influence on the product. Actions that were planned for the first iteration included finding open bugs[6], making significant changes in order to fix the issue, and following through the inclusion of the change into the main version.

[5]A leadership structure based on that contributors with more contributions and experience have a larger say in decisions. [8]

[6]A "bug" means a defect report or issue that needs attention in the source code, either by changing the source code, or by dismissing the defect/issue with some good reason. That a bug is "open" means that noone has claimed it as their responsibility.

Finding bugs that were easy to work with was harder than expected. Three bugs were addressed in this phase, but with regard to the number of code lines the contribution was small.

From the interactions in this process, Netbeans was found not to significantly differ from meritocratic hierarchies in other OS communities. However, one surprising discovery was that most contributors seemed to be employees of the maintainer organization.

3.4 Evaluation

Analysis of data was done with a pre-defined plan that involved considering the observations in context of a theory. Davison et al. state that theory provides a basis for delineating the scope of data collection and analysis [6]. Assessing findings in a broader context also increases confidence in the results.

Thought maps and categorization of the findings was also used to manage own interpretations in a more creative way. Distinguishing between what are facts and what are judgements must be done to allow any reader to make up their own assessments and interpretations [6].

Tangible results that were found from the project included findings in the Netbeans community that there may be problems in attracting a large volunteer workforce. However, the Netbeans project does implement the Open Source model well, and values all outside contribution. Further investigation will be needed to see if these tendencies are universal to other OSS projects where commercial organizations are maintainers.

The project described here has been evaluated by a professional who works in a software company and who has long experience with open source projects both as member of his organization and as volunteer. This professional is positive to the results of the project. Both during the formal evaluation meeting and during informal conversations, this and other professionals recognize the importance of the project in particular and the assigment in general as a way to educate software engineers and to accumulate knowledge about open source projects. He criticises the nature of the research questions which are not enough related to the software engineering domain.

During the project, the time schedule was found to be unproblematic. However, more time during the project participation part would be preferable. Joining a OSS project, getting familiarised with the project artifacts, while also contributing to it, takes considerable effort.

3.5 Reflection

Actions for the second iteration would focus on participating to one module within Netbeans, and looking closer at the artifacts surrounding it. The "JavaCVS" module was selected. Only one bug lead to a successful resolution during this iteration, which incidentally was unrelated to the JavaCVS module. What was learnt from this iteration, was support for the notion that few participants outside of Netbeans were active.

After this iteration, the action research cycle was ended, as the time constraints were exhausted, and sufficient information to discuss research questions had been collected.

In retrospect, there are many ways in which participation to OSS for education can be made smoother. First, focusing on OSS as a social discipline can help the researcher to get access to the project artifacts, and contribute to valuable knowledge both about culture and product. Following the Action Research discipline, the researcher should go to length to collaborate with other people during the project execution, for instance through discussing ideas and technical solutions in mailing lists or newsgroups. In the study reported here, however, collaboration was not easy and therefore not practiced much.

Second, a good recommendation is to focus research on one restricted domain, like a particular module or functional area. While this was not extensively practiced in this case, it is beneficial to commit to one particular role in order to get a more likely "open source situation".

At the end of the project there were no problems concluding the research activities. Participating in a social system with other individuals always carries the danger of "going native", which should be avoided [6]. This was not perceived to be a problem in this case.

Netbeans is evaluated to be a good choice as it is maintained by a larger software company, Sun Microsystems, that also invests significant resources to sustain it. Netbeans is a development tool that is used to aid in the development of Java-based applications.

Experience from the project, however, show that the project selection criteria may not have been optimal. The following was noted after the completion of the project:

Maturity: Selecting an OSS project that has a low level of maturity may have the disadvantage of being significantly different from an ideally run OSS project. However, if an OSS project is mature, well-tested, and close to a release, much of the remaining tasks will be polish. If the goal of the project is to contribute to an OSS project, the researcher should at least be aware of possible difficulties. In this case with Netbeans, contributing to it was difficult due to the difficulty of understanding complex bug reports.

Size of project: Larger OSS projects may suffer from awareness problems. Entering an open source project consisting of thousands of source files requires either excellent skill, or good documentation.

4. DISCUSSION AND CONCLUSIONS

The final goal of this work is providing guidelines on how to exploit open source software for education and empirical purposes. At the time of writing we can provide two examples of projects that exploit open source software for education and empirical purposes [21] [16]. The first [21] was a 1000 hour final master's project where two students participated. The second project which is described in Section 3 was a 500 hour project with one student. There is an ongoing third project (1000 hour) in which the same student who participated to the case reported in Section 3 is involved.

There are four main axes around which to organize an evaluation of our goal:

1. Research questions:
   Working on the research questions is a time-consuming task that required a good understanding of the domain. Here there is a tradeoff between learning and research issues. While students appreciate the freedom of the assignment as a positive learning experience, it is more effective from a research perspective to provide students with predefined research questions.

These can be taken from related literature or from previous research projects the teacher/researcher has been working with. There is a relationship between research questions and projects. For example in the case reported in this paper, the research questions are about the interaction between professionals and volunteers in the OSS projects and this makes it necessary to select a project in which commercial actors play a significant role. From the point of view of the industrial professional who evaluated our work and also from discussions with other researchers, it was found that it is better not let students to choose research questions.

2. Research methods:
   Action research has worked well to balance student's learning, and the output of the study. A deep understanding of the problem itself is not necessary before intervention in projects.

   The effort necessary to contribute to an open source project should not be underestimated. A common problem for both projects, was that the students started with too ambitious goals, and therefore may have run into some difficulties.

   The different iterations used by the researcher to evaluate the problem may take considerable time and energy. The effect of this, is that learning about open source development in general, will be a continuous process throughout the entire intervention period.

   For the sake of presentation and discussion we have presented our case according to the five phases in action research (diagnosis, planning, intervention, evaluation, and reflection). We are still discussing how the different phases overlap with each other. Take for example the project selection phase which we regard as a sub-phase of diagnosis. In other action research projects, the choice of the projects to work with may happen before the whole research process is started. The same is valid for research questions (or goals) which can be less open to be decided inside the AR cycle than in our case. Evaluation and reflection are two related phases that could be merged together.

3. Literature: The open source literature ([19, 24, 17, 9, 2, 3, 10, 22, 23, 15]) is oriented toward the social side of open source software. This influenced the choice of the research questions which in turn influenced the size and the nature of the project.

   A literature review must be performed to update the content of the supporting course and provide a theoretical background that reflects the evolution of the OSS research field. This is an ambitious task as OSS research efforts have been published in the main software engineering conferences and journals in the last few years. There is also an increasing number of books on this subject that have been published in the last couple of years. This makes the task of maintaining a map of open source literature a challenging one.

4. Choice of the open source project: In communicating with OSS projects, problems in the last project included entry difficulties and problems handling the size of the project. The technical competence needed to contribute was here higher than anticipated. More participation in mailing lists and newsgroups may have been helpful to respond to this problem.

   For students who already engage in open source development, encouraging them to choose project they are already familiar with may reduce the time necessary to get acquaint with source code and the development process.

Guidelines for using OSS in education should stress that it is a social and complex discipline. Learning both empirical methods and getting an introduction to the open source is difficult. As mentioned, we have an ongoing project that is run according to the same principles in the case reported here and we plan to propose the same kind of projects, both 500 hour and 1000 hour projects in the next academic year.

Concerning the research method, we are satisfied with the use of action research and we believe that this paper is a valuable description of the exploitation of this method. Concerning the research questions there will always be a phase in which the student and the supervising researcher select new ones starting from consolidated questions in the general literature or provided by this family of projects. The choice of the open source project is an interesting topic of discussion. While it is in the interest of the teacher/research to decide the project in which the student work, we have to keep in mind that one of the principles of OSS participation is motivation and interest. Students who choose these projects are students who love participating in open source projects and that may have good experience with a specific issue. By letting the student participate in this project, the teacher/researcher could get valuable insights in the specific project.

The perspectives of the industry here is similar to that of the researcher in that industrial actors are naturally interested in letting students work on the OSS projects they support to increase activities in these projects. By replicating these kinds of projects we aim to develop a characterization of both open source projects and research issues.

5. REFERENCES

[1] Rudolf Andersen, Reidar Conradi, John Krogstie, Guttorm Sindre, and Arne Sølvberg. Project Courses at the NTH: 20 years of Experience. In *J. L. Diaz-Herrera (ed.): "7th Conference on Software Engineering Education (CSEE'7)"*, pages 177–188, San Antonio, USA, 5–7 Jan. 1994. Springer Verlag LNCS 750.

[2] Magnus Bergquist and Jan Ljungberg. The power of gifts: organizing social relationships in open source communities. *Information Systems Journal*, 11:305–321, October 2001.

[3] Andrea Bonaccorsi and Christina Rossi. Why open source software can succeed. *Research Policy*, 32(7):1243–1258, July 2002.

[4] Jeffrey Carver, Maria Letizia Jaccheri, Sandro Morasca, and Forrest Shull. Issues in using students in empirical studies in software engineering education. In *IEEE METRICS*, pages 239–, 2003.

[5] Claes Wohlin and Per Runeson and Martin Hőst and Magnus C. Ohlsson and Bjőrn Regnell and Anders Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, 2000.

[6] Robert M. Davison, Maris G. Martinsons, and N. Kock. Principles of Canonical Action Research. *Information System Journal*, 14(1):65–86, 2004.

[7] Jianjun Deng, Tilman Seifert, and Sascha Vogel. Towards a product model of open source software in a commercial environment. In Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors, *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 31–37. International Conference on Software Engineering, May 2003.

[8] Roy T. Fielding. Shared leadership in the apache project. *Communications of the ACM*, 42(4):42–43, April 1999.

[9] Nicolas Gold, Claire Knight, Andrew Mohan, and Malcolm Munro. Understanding service-oriented software. *IEEE Software*, 21(2):71–77, March 2004.

[10] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81, New York, NY, USA, 2004. ACM Press.

[11] M. Letizia Jaccheri. Software quality and software process improvement course based on interaction with the local software industry. *Computer Applications in Engineering Education*, 9(4):265–272, 2001.

[12] Chris Jensen and Walt Scacchi. Collaboration, leadership, control, and conflict negotiation in the netbeans.org community. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS 05)*, page 196b, 2005.

[13] Timothy C. Lethbridge. The Relevance of Software Education: A survey and some Recommendations. *Annals of Software Engineering*, 6:91–110, 1998.

[14] Wolfgang Maass. Inside an open source software community: Empirical analysis on individual and group level. pages 65–69. International Conference on Software Engineering, May 2004.

[15] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[16] Håvard Mork. Leadership in hybrid commercial-open source software development. Directed study, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, December 2005.

[17] Tim O'Reilly. Lessons from open source software development. *Communications of the ACM*, 42(4):32–37, April 1999.

[18] Daniel Port and David Klappholz. Empirical research in the software engineering classroom. In *CSEE&T*, pages 132–137, 2004.

[19] Walt Scacchi. Free and open source development pratices in the game community. *IEEE Software*, 21(1):59–66, February 2004.

[20] Mary Shaw. Software engineering education: a roadmap. In *ICSE - Future of SE Track*, pages 371–380, 2000.

[21] Sigurd Tjøstheim and Morten Tokle. Acceptance of new developers in oss projects. Master's thesis, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, June 2003.

[22] Eric von Hippel and Georg von Krogh. Open source software and the 'private–collective' innovation model: Issues for organizational science. *Organization Science*, 14(2):209–223, March 2003.

[23] Georg von Krogh, Sebastian Spaeth, and Karim R. Lakhani. Community, joining, and specialization in open source software innovation: A case study. *Research Policy*, 32:1217–1241, July 2002.

[24] Huaiqing Wang and Chen Wang. Open source software adoption: A status report. *IEEE Software*, 18(2):90–95, March 2001.