The User-Developer Convergence

Innovation and Software Systems Development in the Apache Project

Thomas Østerlie

The User-Developer Convergence: Innovation and Software Systems Development in the Apache Project

by Thomas Østerlie

Revision History

Revision 1.0 November 14, 2002
Original master thesis.

Table of Contents

Chapter 1. Introduction

Apache is the most widely used Web server on the Internet according to Netcraft's Web server survey (http://www.netcraft.com). The survey shows Apache has been the most widely used Web server since early 1996, and remains so with a wide margin. Of the 35,114,328 Web sites under survey in October 2002, 60.54% are running Apache. The second most used Web server is software giant Microsoft's Internet Information Server at 28.89%. Judging by its success, Apache must be a real feat of engineering. Oddly enough, the Web server is developed by a group of volunteer programmers distributed across the United States and Europe. There has been no central project management, a mailing list the only means of communicating and synchronizing the development effort.

We are taught that software systems development is an engineering discipline, akin to building bridges or architecting buildings. Certain elements must be in place for a software development project to succeed. As Booch, Jacobson and Rumbaugh writes:

> There is a belief held by some that professional enterprises should be organized around the skills of highly trained individuals. They know the work to be done and just do it! … This belief is mistaken in most cases, and badly mistaken in the case of software development. … [D]evelopers need organizational guidance, which … we refer to as the "software development process". (Booch et al. 1999, p. XVII)

The problem domain must be charted, and the system requirements formulated. The system requirements detail the software system's functional requirements. By identifying the software system's functional requirements, we are ensured the right system is developed. Once this initial survey is done, it is about developing the system right. A detailed plan is laid down to show how to build the system. Charts and diagrams are drawn to show the system's architecture. Upon completion of these, it is time for the programmers to code the system and make it into runnable software. Surveying, planning, building; the steps required for any engineering activity to succeed.

Looking at the Apache Web server development, two of these elements are missing: surveying and planning. Apart from a standard describing the network protocol used by the Web server to communicate with its clients, there are no other written requirements. This does not mean the Web server is no more than a piece of software processing HTTP requests. It is a file server, returning requested files to its clients. Even such a fundamental piece of functionality as where the server is to look for the requested files, is non-existent in any standard or requirements document. Apart from returning a syntactically correct HTTP error response to the client, how are errors handled internally in the Apache Web server? Not to mention all the added functionality such as access control, database interfaces, virtual hosting, just mention a few. There was never an explicit plan for building these; no initial surveying done, no charts produced, the feature is just implemented.

“Distributed engineering and development depends upon careful planning, coordination, and supervision” (Moon and Sproull 2000). The work process used in developing the Apache Web server is described in a single, short Web page (Hartill and Fielding 1995). This is an in-depth description of the project’s decision-making process. It is about how to determine whether a new feature or a bug fix goes into the next release or not. A ballot is held to determine what new features and bug fixes to include with the upcoming release. The vote of every participant on the mailing list counts. The timing of the ballot depends on someone feeling like releasing a new version of the Web server. Anybody is free to declare a ballot, and prepare the new release. There is no supervision, although it is possible for any developer to veto a change or new feature. Careful planning, none existent. The only means of coordination is communication pr. e-mail on an electronic mailing list.

Yet, the Apache developers succeed with their endeavors. After a year or so after starting working on it, theirs is the most widely used Web server on the Internet. In 1996 it is adopted by the Internet Engineering Task Force as a reference implementation of the new release of the Hypertext Transfer Protocol, version `1.1`. Even though doing almost everything wrong from an engineering perspective, the Apache developers are obviously succeeding with their efforts. How come?

Can it be that software systems development is more than just an engineering activity? Can it be that other factors play an equally important role in developing software? What exactly is it that the Apache developers are doing, and how can it affect the way we understand software systems development?

During the mid- and later parts of the 1990s, both Microsoft and Netscape funneled enormous sums into developing the emerging Web technology. With no commercial backing until IBM starts showing interest in 1998, Apache is one of the prime actors in setting the agenda for innovating Web technology. Features that are taken for granted today, were not even thought of when Tim Berners-Lee developed the first Web server. Even though money is spent on corporate research, innovations spring from the volunteer efforts. The PHP scripting language, for instance, has its roots as an embedded scripting language for server-side processing of Web pages on Apache. Microsoft launches its competing technology, Active Server Pages, a year after PHP’s launch in 1995. PHP was originally written by a young programmer from Canada for tracking access to his resume (http://www.php.net/manual/en/history.php). From this it grew into one of today’s most widely used embedded scripting languages for server-side processing of Web pages.

The story of PHP is not unique in the history of the Apache Web server. Instead it seems to be more of the norm. Several of the technologies that today make up the World Wide Web, originated as part of the Apache Web server. What is particular to the team developing Apache? What makes the Apache developers innovate? Maybe innovation is more than just pure analytic, cranial work? It may seem that other factors influence the evolution of technology. An interesting fact with the Apache developers is that all developers are also users of the Web server. They are webmasters or even professional consultants selling services on the World Wide Web. Can the particular relationship between user and developer play a role in innovating?

Towards the end of the 1990s software created by the hacker underground emerged as viable alternatives to its commercial counter-parts. Linux and Apache perhaps are the most renowned and widely used software systems to emerge from hacker communities. There is mythical ring to the name *hacker*. Popular scientific literature (Levy 1984) has had a tendency of romantically portraying the hacker as a techno-anarchist with an uncanny ability to write ground-breaking software. Some attribute the hacker communities with significant innovative abilities (DiBona et al. 1999). Others accuse the hackers of simply chasing taillights, imitating software already developed by commercial actors (Valloppolli 1998). During the worst part of the 1990's IT craze, hacking was even claimed the foundation of a new software economy (Young 1999). But what is hacking, and how does it relate to software systems development? What is it with hacking that makes it different from software engineering, and how does that relate to innovation?

Attempts have been made to prescribe hacking as an approach to software systems development (Browne 1998) (Hannemyr 1999) (Raymond 1998). Hallmarks of hacking have been identified. Eric Raymond (1998) presents a list of enabling conditions for to succeed with developing large-scale projects through hacking. His work is based on personal experience. Raymond is one of the hacker community's most outspoken representatives. His work describes only the trappings of hacking, so the question still remains: what is hacking? Maybe the a way to understand what hacking is, is by looking at what a hacker does? Would it be possible for such an approach to uncover new insight into hacking as a form of software systems development? If it is so that hacking has produced great innovations, would it maybe be possible to learn something about innovation as well by studying hacking?

With the Internet's rise from academic obscurity in the early 1990s to mass use by the middle of the decade, tele-commuting and computer supported distributed work became the order of the day. A wide range of software for computer supported cooperative work was developed to assist the new distributed form of working. Large corporations developed and installed elaborate groupware systems perceiving massive gains in productivity from the introduction of technology (Monteiro and Hepsøe 2000). Without little or no work apart from introducing the technology, it was believed that groupware and the Internet would change the way organizations cooperated and exchanged expertise (Orlikowski 1992).

The Apache project uses only an electronic mailing list and a file server in its collaborative work. Yet, it seems to suffice. The mailing list is archived for future reference. That's it when it comes to saving organizational knowledge electronically. No elaborate groupware seems to be required for a relatively large, distributed collaboration effort to succeed. What lessons can be learnt from the Apache project about distributed collaborative work? Aren't the Apache developers exchanging information and knowledge? Can it be that there are other elements other than technology that need to be in place in order for distributed, collaborative work to succeed?

This thesis is an effort to understand the software systems development practices of the Apache hacker community based on a case study of the Apache project. Is there something in their seemingly *ad hoc* approach that can shed some new light on how better to develop software? Is there something in their approach that makes this way of developing software more ideally suited for innovation? And in what way does it enable us to say something about distributed computer supported collaborative work?

Chapter 2. Software systems development

Software systems development can be understood as the effort undertaken to create software for computer-based systems. There are a number of approaches to developing software. What exactly software systems development actually is depends on the context. Whether developing software for personal use or enterprise systems, the core activity is is nevertheless the same: creating software for computer-based systems. The difference lies in scope, complexity, and context. Developing software used in landing aeroplanes requires a lot more accountability than developing software for automating trivial tasks. Developing an enterprise system involving numerous departments and employing scores of developers requires a whole other type of management technique than managing a small team of a handful developers. Different approaches have been devised to meet the different environments software is developed in and for.

Software systems development as a discipline has evolved since its infancy in the mid-20th century. In the early days of computing, software costs comprised a small percentage of overall computer-based system cost. Throughout the late 1960s and the 1970s software became an increasingly more important part of such systems. The application of computer-based systems grew. With growth came added functionality, and with added functionality complexity increased. Scientists and practitioners have tried to devise new ways to meet the challenges of building large-scale computer-based systems (Pressman 1996). Yet, “an increasing number of practitioners are beginning to realize that old approaches are not appropriate any more” (Miller 1992, p. 93).

A short time-line

The first programmable computers were set to use in the mid-twentieth century. In the early stage of computing computers were used as large programmable calculators, set to solve complex mathematical problems. Hardware was initially the limiting factor, but the hardware was rapidly improving, though. With improved hardware the appliance of software grew, leading to increased complexity in software. Already in the early 1960s computer professionals warned about these problems. “A combination of increasing complexity of systems and the relative inexperience of systems development staff led to late deliveries of systems, escalating costs and failed software projects” (Friedman 1989, p. 99). Another pertinent issue was the increased cost and problems with maintaining existing software systems. The limiting factor was shifting from software to hardware. Some came to call this shift of emphasis the *software crisis*.

By the late 1960s the problems cause by this shift were becoming apparent. The computer industry as a whole increased focus on how they developed software. A turning point came at the Garmisch Conference of 1968. Addressing these problems, the conference participants agreed that the solution would be to apply tried and tested methods of traditional engineering to software development. Calling this *software engineering*, they recommended that by standardizing the program structure through modularization, and to order the systems development process, their problems would be solved. "These recommendations would increase the observability and measurability of systems development and thereby increase management control" (Friedman 1989, p. 106).

In Fredrick Brooks (1975) argues that software engineering was not the silver bullet to solve the problems facing software systems developers. During the 1970s it was becoming apparent that software engineering had its limitations. There were still problems left to be solved. Different responses to this failure were suggested. One of them, as drawn up by Claudio Ciborra, explains the problem accordingly:

> Information systems development often does not lead to a more effective and human organization . . . because systems designers do not apply structured methods or organization . . . (Ciborra 1987, p. 2).

The answer to the problem is not Ciborra's but rather his recounting of the arguments of software engineering's proponents. For these the response to the short-comings was to continue the quest for a better methodology. During the late 1970s and early 1980s a new methodology emerged—*object-orientation*. With its roots in a programming paradigm from the early 1970s of collecting data and their operations into objects, practitioners saw the possibility of transferring the basic principles of object-orient programming into the domain of software engineering. By the 1990s this approach would be embraced by the computer industry as a whole as the silver bullet to solve all problems of software engineering. While it is apparent that object-orientation did help bridge the gap between systems modeling and realization and proved itself an invaluable tool for conceptualizing software systems, did it really address the underlying problems of classic software engineering or was it simply yet another methodology in a long line of succession?

As Miller observed in 1992, despite the efforts of the software engineering community "an increasing number of practitioners are beginning to realize that old approaches are not appropriate any more" (Miller 1992, p. 93). Since the early 1970s, a group of researchers and practitioners have argued that neither software nor hardware are the limiting factors in software systems development. They argue the short-comings are human. Software engineering has eased issues like schedule slippage, bugs, and failure to stay within budget, but do they address the source of the problem? As early as the late 1960s and early 1970s software systems developers note the discrepancy between users requirements and software systems

(Boguslaw 1965) (Mumford 1972). It is argued that software "systems has to be useful in terms of allowing users to do their jobs better" (Friedman 1989, p. 175). Software constraints are no longer the sole limiting factor; user relations constraints are also a limitation. Not only computer professionals admitted human-computer interaction was poor, studies shows developers have problems understanding the users' needs and requirements for software systems.

For those subscribing to this new view of software systems development it was no longer a question of producing the software system right, but producing the right software system to meet its users' needs. This strain of software systems development is sometimes called the *alternative approach*. The alternative approach springs forth from as varied fields as linguistics, psychology, and sociology. Their only common denominator is that they seek alternative ways to those of software engineering. An implication of the shift of focus from producing the software systems right to producing the right software was a shift of attention towards the users' needs and requirements came into focus. With this shift came an appreciation that there were conflicting interests between participants in the software systems development process—especially users and management (Bansler 1989).

With its roots in technical communities of the 1960s, hacking has been a strain of software systems development running in parallel with software engineering for the past 30 odd years. Hacking is not to be confused with its malicious cousin, cracking, which aims at breaking into software systems. An entire mythology has been spun around hacking over the past three decades. The first written accounts on hacking is the Jargon File, an inside view of how the hacker community wants to view itself. The Jargon file was later published as *The New Hacker's Dictionary* (Raymond 1998). The Jargon File was a collection of jargon from early technical communities in the United States, the first hacker communities. The Jargon File became, quite typical of the culture, a shared repository of technical slang being updated by a great number of people during its almost thirty year lifespan.

The hacker is "a person who enjoys exploring the details of programmable systems and how to stretch their capabilities" (Raymond 1998, p. 233). The *hack* is "an incredibly good, and perhaps very time-consuming, piece of work that produces exactly what is needed" (Raymond 1998, p. 231). Hacking is consequently the activity a hacker undertakes to produce a hack. Hackers like to see themselves as computer experts pushing the limits of technology. Writing about the early hackers by MIT's AI Lab, Steven Levy identified six principles he coined *the hacker ethic*, of which the the hands-on imperative, the principle that all information should be free, and to distrust authority and promote decentralization are those that rings through all literature on hacking.

At the core of hacking lies key values as openness, sharing, freedom and cooperation. Its practitioners considered hacking a community based approach to software systems development. As an approach to software systems development, hacking has produced viable systems in wide-spread use. Examples include the

Unix operating system, the Internet standards, the GNU project, and more recently the Linux operating system and Apache Web server.

Software engineering

What does it mean to apply tried and tested principles of engineering to software systems development? The engineering terminology applied to software systems development is used with different connotations depending on writer and context. The differences are often sublime, and may lead to a confusion of terminology. *The Oxford Concise Dictionary* defines engineering as “the application of mathematical and scientific principles to practical ends, as in the design construction, and operation of economical and efficient structures, equipment, and systems” (Allen 1991, p. 388). Traditionally, the engineering disciplines divide work into distinct phases, each phase based upon the results of the preceding phases. Plans are made before building, and planning is often based on a preceding analysis of the task at hand. The charts and diagrams drawn during planning is used in construction.

Three terms are central to software engineering: process model, method and notation/technique. I use the term process model to indicate an overview of the development effort’s life-cycle. I will sometimes refer to the process model as simply the model, at other times as the process model. The nuance between the two will be further explored later on. The process model is a description of the development effort’s flow of work, which phases it is to be split into, and the activities performed in each phase. The process model describes the entire life-cycle of the development effort, from the inception of the idea to the actual realization of the end product. It encompasses the methods used to analyze, design, and test computer software, the management techniques associated with the control and monitoring of software projects. The process model prescribes what activities is to be done and when in the development project’s life cycle to undertake the activity. How to perform the activities is the methods’ domain. Object-oriented analysis and design is a method. The Unified Modeling Language is a notation, or technique, that may be used with OOAD. Booch is another technique.

The basic idea of any software engineering process is to first analyse the software requirements. Requirements are often described in natural language. The specifications are then codified into a design. Writes Martin Fowler: “The analogy is [that] design is an engineering drawing” (1999, p. 15). The design is often split in two: the top-level design and the detailed design. Top-level design is sometimes called the software architecture. Booch, Jacobson and Rumbaugh calls the architecture design a common vision for the development effort (1999). It usually describes the general mechanics of the software system. The detailed design is more of a work plan to base the coding on. Coding, that is building the software system upon the plans laid down earlier in the development effort, is incidentally most commonly disregarded by software engineering.

To give a better view on software engineering, let’s have a closer look at two process models, Jackson’s software development method and the Unified Software Development Process.

Jackson’s software development

British computer scientists Michael Jackson and John Cameron launched *Jackson System Development* (JSD) in 1983 with the book *Software Design* . The method is a continuation of Jackson’s earlier work *Jackson Structured Programming* (Jackson 1975). JSD is a life-cycle process, meaning that it aims at covering the entire development process from mapping the part of the real world which is taken into account when developing the software to implementing the software system. JSD uses a precise notation for specification and implementation. The transition to implementation is not just a detailed re-statement of the specification, but rather the result of applying transformations to the initial specification.

By identifying and linking processes, the software system’s architecture is laid down. Processes are linked in one of two ways: by asynchronous data stream connections, or by state vectors. A state vector allows one process to see the internal state of another process without the need to communicate with it. The architecture models “the reality with which the [software] system is concerned, the reality which furnishes its [the software system’s] subject matter” (Jackson 1983). The goal of modeling is to formalize the relations between real-life processes and processes in the software system.

The focus of design in JSD is the data flow. A good design is one that incorporates this natural flow. Processes themselves are categorized naturally into a few distinct types, an a top-down design approach that targets these types will result in a design that is easy to realize.

Having obtained the design, its implementation must be accomplished in a systemic manner. With a message-based concurrency model within the implementation language this is much easier, as design processes and buffered data flows can all be coded as program processes. Unfortunately, this can lead to a proliferation of processes and a very inefficient implementation.

Most programming languages are not amendable to JSD’s transformation techniques. The approach advocated within JSD is a transformation known as inversion. In this, a design process is replaced by a procedure with a single scheduler process controlling the execution of a collection of these procedures; that is, rather than a pipeline of five processes, the scheduler would call five procedures each time a data item appeared (if the five processes were identical then obviously there would only be one procedure, which would be called five times).

JSD is a variation on the waterfall model. Jackson details six phases in developing software, each building on the output of the previous step(s). The first two steps—entity action step and entity structure step—are aimed at capturing the software requirements. Top-level design is the result of the third step, the initial model step. Two aspects of detailed design—user functions and timing—is developed during steps four and five. In the sixth and final step are the plans developed used to build the software. This step is called the implementation step. Once the software is implemented, it is ready to be delivered.

The unified software development process

Object-orientation is founded on Kristen Nygaard and Ole Dahl's work on the Simula programming language in the early to late 1960s (Holmevik 1995). As such object orientation has its roots in the programming side of software systems development. During the early 1990s, the object-orientation community saw something which in posterity is dubbed the method wars. By the mid-1990s there were numerous competing object-oriented approaches to software systems development, with no indication of standardizing notation nor techniques. Efforts were made by the independent standardization organization, the Open Management Group, an invitation that was promptly refused by all camps. In an unexpected turn of events, three of the most prominent camps joined up to produce the Unified Software Software Development Process. Declaring victory in the methods war, they started standardizing the way object-oriented software was to be developed (Fowler 1999).

The Unified Software Development Process, the unified process for short, is a software engineering method for developing object-oriented software. "A process defines *who* is doing *what when* and *how* to reach that goal" (Booch et al. 1999, p. xviii). This is the Unified Software Development Process' creators' goal with their process model. Their process is made up of a number of work flows. A work flow is a sequence of activities that produces a result of observable value. The work flow says something about when to start one or more activities. The activity lays down who is to produce what. The result of an activity is one or more artefacts. An artefact is an ephemeral entity, used to describe just about any tangible product of an activity. It may be a diagram, a document, source code, executables, just to mention a few concrete examples. The worker is responsible for producing the artefact. The worker refers to the roles that define how the individual participants in the development effort should do the work. Examples of workers are system analyst, designer, test designer, to mention a few.

The unified process provides the general framework for a development process. Knowing that no single process fits all sizes, the unified process is centered around the development case. The development case is a tailored process model for an individual project. In fact, one of the unified process' central work flows is the environment work flow. Its purpose is to support the development organization with both processes and tools. The process aspect is taken case of by the process configuration activity, which artifact is the development case.

While independent of technique, the unified process is developed for use with object-oriented development. It uses the Unified Modeling Language for notation. It is used in modeling everything from the development process, to business modeling and the implementation diagrams used in coding the software. The unified process/UML is tightly interwoven, and while the UML is a standard under the Open Management Group's control, it is tailored to fit Rational Corporation's Unified Software Development Process, or the Rational Unified Process as it is also called.

A large part of the unified process focuses on modeling. The process is described as both architecture-centric and use-case driven. Requirements are captured with use-cases. During the requirements work flow, the software system's requirements are captured and described, or using the unified process' terminology: the problem is understood and modeled. The use-case is used to formulate a model of the solution to this problem, it is a graphical representation of a set of requirements. The use-case is a sequence of actions a system performs that yields an observable result of value to an actor. Actor is in this context used to denote a non-specific entity that makes use of the software. The actor may be a user, but it may also be another software system or possibly a hardware system.

The unified process' creators argues the importance of architecture. Architecture helps understanding the software system, it organizes development and fosters reuse. "The software architecture embodies the most significant static and dynamic aspects of the system . . . Architecture is a view of the whole design with the important characteristics made more visible by leaving details aside" (Booch et al. 1999, p. 6). Different kinds of notations are used to model the software system during the analysis and design work flow. In the end the idea is to have as complete drawings as possible to base the implementation on during the construction work flow.

Process models based on the waterfall principle has shown too rigid for software systems development. The unified process solves this by being iterative. This is a common solution to the problems with waterfall processes, and nothing particular to the unified process. Being iterative means that instead of developing the software in one go, the development effort is broken into a succession of small waterfall projects. Upon having completed all stages—or work flows in the unified process—of the waterfall, the process is started all over again from the beginning. Each run is called an iteration. This way software is built incrementally. The advantage of iterative development is that it is presumable more flexible to change. By repeating the requirements gathering, the process is able to cope with a changing environment which the software will be deployed in. It also allows for knowledge created during the development process to be fed back into the development effort. So while each iteration is sequential like waterfall models, the iterative process is different as the work flows are revisited again and again through the development effort’s life cycle.

As the development effort passes through a series of iterations, the process model increases and decreases emphasis on different work flows. The analysis and design work flow is more important in early stages of the development effort, while focus on construction increases with each iteration. In the end, when the software is complete focus increases on the deployment work flow, which hasn’t had any focus in earlier iterations.

Discussion

What are the differences and what are the similarities between Jackson Development Process and the Unified Software Development Process? How and in what way do they both belong to the tradition of software engineering?

If engineering is “the application of mathematical and scientific principles to practical ends” (Allen 1991, p. 388) and software engineering is applying engineering to software systems development, then software engineering’s world view is described by Burrel and Morgan as objectivist. Writes Burrel and Morgan:

> [T]he objectivist applies models and methods derived from the natural sciences to the study of human affairs. The objectivist treats the social world as if it were the natural world. (Burrel and Morgan 1979, p. 7)

Neither JSD nor the unified process provides any techniques other than notation to understand “the part of the real world which [is] taken into account when developing programs” (Floyd 1987, p. 198). Floyd calls this the referent system, as opposed to the software system which is the software that interfaces with the referent system. Both JSD and the unified process apply classification of the real world into entities and processes/classes and actors. The basic assumption is that it is actually possible to classify and translate social processes, the referent system, into charts and diagrams. The world is classified, and relationships are drawn between the classes—inheritance, aggregation—and then the classes are drawn up in a hierarchy. Subjectivity is not an issue, as the referent system is inherently classifiable.

Christiane Floyd draws up a dichotomy between what she calls the *product-oriented* and *process-oriented* perspective on software systems development. Software systems development traditionally views “software as a product standing on its own, consisting of a set of programs and related defining texts” (Floyd 1987, p. 194). She calls this the product-oriented perspective. From this point of view the world is static, having two states: before and after the software system is introduced. “The process-oriented perspective, on the other hand, views software in connection with human learning, work and communication, taking place in an evolving world with changing needs” (Floyd 1987, p. 194). Floyd’s argument is quickly summarized by the software development cliche: It’s no longer a matter of developing the system right, but developing the right system. To Floyd the product-oriented perspective is concerned with developing the system right, while the process-oriented perspective looks at developing the right system.

It is easy to draw an equality between the product-oriented view and the objectivist perspectives of software engineering. Especially since Floyd says the product-oriented perspective chooses the referent system with a view to successfully develope the software system, and that programs are derived by formalized procedures starting from an abstract specification. The abstract specification being the architecture and models both JSD and the unified process emphasizes. The picture, however, is more nuanced.

The product-oriented view regards the referent system as informationally closed and essentially static, having basically two states: before and after introducing the software system. Methods are generally applicable, well-defined and context independent. The process-oriented view, on the other hand, views software systems development as a process of learning and communication. Herein lies the difference between JSD and the unified process, and the difference between a model and a process model. JSD shares the product-oriented perspective on the referent system. That it is shaped on the waterfall model reflects the view that the referent system has two states: before and after the software system is introduced. The JSD approach to software systems development is a context free description of the work to be undertaken. Iterative software development exhibits the exact opposite view, seeing the development effort as a process of learning and communication. Thereby the use of process model to describe the development effort’s life cycle.

This difference isn’t unique to JSD and the unified process. It shows a divide that is apparent within software engineering. Many recent additions to software engineering, like the crystal methodologies (Cockburn 2002) and Coad and De Luca’s feature-driven development (Palmer 2002) in addition to the unified process, embrace the learning aspect of the process-oriented perspective by use of iterations. Older development methods, like the Jackson Software Development, are modeled on the waterfall process. Despite their difference in view on the software’s role in the real world, both JSD and the unified process are based on the basic principle of software engineering: applying mathematic and scientific methods to building software.

Participatory design

Robey and Markus (1984) shows that developing software systems is not necessarily a rational process intended to achieve identifiable and agreed upon goals like an objectivist would argue, but rather “a process that can be interpreted as rituals which enable actors to remain overtly rational while negotiating to achieve private interests” (Robey and Markus 1984, p. 5). In the article *System’s Development in Scandinavia: Three Theoretical Schools* Jørgen Bansler (1989) shows that software systems development need not be viewed as an engineering discipline. It is instead viewed as a component in larger socio-economic contexts.

“Participatory design is about involving users in the creation of computer system” (Miller 1992, p. 93). There are usually three arguments why participatory design will improve software systems The first two are practical. Participator design improves the knowledge upon which the system is built, and it enables people to develop realistic expectations while reducing the resistance to change (Bjerknes 1995). Miller claims user participation makes “sure that the end product actually serves their [the users’] needs” (Miller 1992, p. 93). The third argument is culturally and politically biased. A goal in Scandinavian research projects in software system development has been to increase working life democracy. Participator design has been explored to achieve this goal. Participatory design is traditionally justified by three arguments. It is believed participator design will increase working place democracy by giving the members of an organization the right to participate in decisions that are likely to affect their work.

Participatory design isn’t a unified approach to software systems development, but rather a family of approaches subscribing to same goals and ideals of user participation. There are different ways this is achieved.

The socio-technical approach

Miller argues “participatory design starts from the insight that the people on the front-line ... are not only in an excellent position to know what works and what doesn’t work in the current set up, but also what might be done to improve the situation” (Miller 1992, p. 95). The theoretical foundations behind this kind of participator design is the so-called socio-technical systems theory. Central to the socio-technical approach is the notion of *job satisfaction* (Bansler 1989). While job satisfaction is an end in its own right, it is also a means for achieving higher productivity within organizations. Instead of a mechanistic approach to software systems development, the socio-technical approach proposes that organizations will be more effective if they are deliberately designed to ensure a high degree of job satisfaction. The argument is that a high degree of job satisfaction leads to increased responsibility, which in turn leads to better productivity. The means to achieve the goal of job satisfaction is participation.

Socio-technical theory treats organizations as two systems, a technical and a social. Disharmony between these systems leads to a suboptimal organization. Each of these systems have their own inner logic, and they operate along different axis. To optimize the social system employees responsibility and commitment is encouraged. Working conditions must meet the psychological and social needs of the employees. The employees must feel they have a meaningful job with variation and autonomy.

In the late 1960s Sweden saw rapid technological and structural change, something both employers and workers considered a problem. Unions were concerned about working conditions, employers about the personnel problems they were experiencing. The Norwegian socio-technical experiments seemed promising. So while several of the Swedish experiments produced interesting ideas on work organization and democratic participation, practical implementation proved more difficult. The employers were satisfied with the socio-technical approach, but not the joint experiments. The employees argued that the conflicting interest between employer and employees became even more manifest when implementing the new methods of working (Ehn 1992).

Trade union based participation

A new group of researchers argued the socio-technical approach didn’t acknowledge the conflict between labor and management. Instead it provided a harmonious view of the organization, arguing for balance between the social and technical systems. This new direction of research, often called the critical tradition, argued that organization were not “cybernetic systems or symbiotic socio-technical systems, but rather *frameworks for conflicts* among various interest groups with unequal power and resources” (Bansler 1989, p. 15). The critical tradition doesn’t reject all aspects of the socio-technical approach. It simply argues that the problems with practical implementation of socio-technical systems reflects the questionable assumption of organizational harmony, rather than short-comings in the socio-technical techniques (Ehn 1992).

The roots of the critical tradition’s work-oriented design can be found in the Scandinavian discussion about the relationship between democracy and work in the 1960s. It was believed that increased working place democracy would increase efficiency and productivity. A large cooperation program was designed and conducted between the Norwegian Federation of Trade Unions and the Norwegian Employers’ Federation, to improve working place democracy. The employers were concerned with rationalization and improved organizational development, while the trade unions wanted to empower the workers. In this climate of cooperation some explicitly stated political projects were carried to strengthen the trade unions, as it was believed stronger unions would contribute to working place democracy. At their annual meeting in 1970, the Norwegian Iron and Metal Workers' Union initiated a project whose objective was to apply workers' perspectives on development and introduction of new technology. Its stated goal was to strengthen the workers' influence with respect to introduction and use of new computer technology. The result of the project included technology agreements, textbooks and vocational training programs on technology (Bjerknes 1995).

However, Pelle Ehn transcends the political importance of participatory design, arguing for the importance of participation in design by both users and developers. While working on a trade union based project, Ehn discovered some approaches were more successful when working with users. "Requirement specifications and system descriptions based on information from interviews were not very successful" (Ehn 1992, p. 62). As it turned out the use of mockups, prototypes, simulations and experimental situations yielded significantly better results. Ehn aknowledges not only the political importance of participatory design, but also the importance of building a shared understanding of the problem domain between software developers and the users. In that Ehn acknowledges that there is more to software systems development than building the right system or building the system right, there is an social element of joint discovery and learning involved as well.

Discussion

Floyd claims the product-oriented view sees the process model as a way to make software systems development less dependent on people (Floyd 1987). A basic assumption behind participatory design and socio-technical software systems development, is that introducting technology will contribute to rationalizing work and de-skilling workers (Bjerknes 1995). This view is based on the notion of a contradiction between capital and labor, a view with basis in Marxist philosophy. The introduction of technology to rationalize the work process, is the central theme of scientific management. Julian Orr writes:

> The basic premise of scientific management is that one can reduce the best way to do a given job to a set of instructions and give those instructions to someone who does not know how to do it independently but who will then be able to do the job by following instructions. In this way, management gets control over their employees, through control of the knowledge necessary to do the job, and can hire cheaper employees since they do not need skilled labor. (Orr 1996, p. 107)

This introduces a new dimension in how software system development is viewed, a dimension of harmony or conflict. Participatory design is explicit on the conflict between those funding the software system, management, and its users, the workers. One of the criteria Hirschheim and Klein uses in classifying approaches to software systems development, is this order-conflict dimension (Hirschheim and Klein

1989). The order view emphasizes a world of order, stability, integration, consensus, and functional coordination. It is a view where everyone within an organization have the same goals, be they upper management or grass root worker. On the other hand is the conflict view that stresses change, conflict, disintegration, and coercion. At its core, their critique of the order view is based on the assumption that there are conflicting interests within an organization, small or large, and that these conflicts have to be acknowledge to understand the organizational problems facing systems development.

That neither of the software engineering process models presented earlier includes any views on the resolving any conflicting interests between users and management is no coincidence. Software engineering doesn't see this as within the scope the software development effort. The unified process explicitly exonerates itself from any conflicting interests, typified by statements such as this: "We think of the architecture of a system as the common vision that all the workers (i.e., developers and other stakeholders) must agree on or at least accept" (Booch et al. 1999, p. 59). Socio-technical critique of software engineering says software engineering optimizes the technical system at the expense of the social system (Bansler 1989). Rather than accepting a conflict between capital and labor, software engineering itself subscribes to the views attributed management. Software engineering views methods as a way to make development less dependent on people (Floyd 1987). In this respect, the methodologists plays on managements side, while software developers subjected to engineering environments can be de-skilled and cheaper, less trained and experienced workers may be hired. Looking at software engineering from a conflict-harmony point of view, it is a tool in the hands of management.

The differing view on the conflict-harmony dimension probably stems from a more fundamental difference in world view. Where software engineering is objectivist view of software systems development, the alternative approaches have a subjectivist world view. "In contrast [to the objectivist position], the subjectivists position denies the appropriateness of natural science methods for studying the social world and seeks to understand the basis of human life by delving into the depths of subjective experience of individuals" (Hirschheim and Klein 1989, p. 1201).

In participatory design both software systems developers and users play other roles than in software engineering. In software engineering the systems developer's primary role is to be an expert in technology, tools and methods of systems analysis and design, and project management. He is there to make systems development more formal and rational (Hirschheim and Klein 1989). What little role the user plays in software engineering, it is largely instrumental to aid the developer in understanding the referential system. Work-Driven Design sees the developer as a "labour partisan" (Hirschheim and Klein 1989, p. 1206). He sides with the workers in achieving increased working place democracy and a higher degree of influence on the organization. Socio-technical development sees the developer as an emancipator or social therapist, where his role is to attain balance between the social and technological systems. Both directions of participatory design sees the user playing a central role in shaping and directing the development effort.

Chapter 3. Knowledge

The previous chapter’s look at software systems development is from a prescriptive point of view. Both software engineering and participatory design aim at saying something about how software systems development is supposed to be conducted. Software engineering’s approach is to apply natural scientific methods to the effort. This objectivist approach aims at reducing the development effort into a series of phases continually transforming work products, and finally ending up with a running software system. With little or no concern about the software system’s context, software engineering’s main concern can be argued as building the software right. Participatory design views software systems in context of use. To participatory design software plays a role in the working place, and is to be considered a tool to improve employees’ working conditions. The view is based in both pragmatics of improving an organization’s productivity and working life politics.

Similar to both software engineering and participatory design is that they proscribe an approach to software systems development based on external factors. Both approaches are rooted in more traditional industrial understanding of work and production. Software engineering looks at developing software as any other semi-automated production, as if developing software is analogous to any other factory production (Cusumano 1989) (Aaen et al., 1997). Participatory design does not even deal with how best to develop software, but sees the running software system merely as a means to an end.

Human and social sciences has concerned themselves with knowledge and understanding for a long time. Lately a shift of focus has come about within computer sciences, building upon the knowledge research within fields such as anthropology (Orr 1996), psychology (Naur 1992), philosophy (Ehn 1988, 1992) and the philosophy of science (Floyd 1987). Instead of looking at software systems development in light of production, the shift of view has been to look at software systems development as an intellectual activity dealing with knowledge and learning. In this chapter I will try to underpin the appropriateness of such a view. The field of knowledge and learning is too broad to fit the scope of this thesis. Instead of an exhaustive treatise on the field, I will therefore limit my topics to those directly applicable in understanding certain aspects of the Apache group’s approach to software systems development.

Software systems development as theory building

At the end of his career Peter Naur writes the article *Programming as Theory Building* where he summarizes his experiences with software systems development by arguing for a view that software systems development is something more than just production of a program and certain texts. He argues that successful software systems development is a question of having an appropriate understanding of both the referent and the software system. His arguments are based on personal experience and presented as two case studies. The first case study looks at making extensions to an existing compiler. The story concerns two software development teams, team A and team B. A has implemented a compiler for language L. Team B wants to implement the compiler for a language L+M, which is language L with minor modifications M. Team B strikes a deal with A, where A agrees to provide B with support in form of full documentation and personal advice. During the design phase, team B comes up with ways to add modifications to the existing compiler. These suggestions are submitted to team A for review. As it turns out, team A finds that in most cases team B does not make use of the existing framework when implementing their changes. This despite the fact that the facilities inherent in the original design are discussed at length in the documentation. Team B's suggestion are instead "based on additions to the structure in form of patches that effectively destroyed its power and simplicity" (Naur 1992, p. 228).

His second case relates to the installations and fault diagnosis of a large real-time system. The people installing the system have been involved with its development over several years. During fault diagnosing they rely on their knowledge of the system and its annotated 200,000 lines of program text. They are unable to conceive how any kind of additional documentation could prove useful to them. Other programmers' groups are responsible for the operation and maintenance of particular installation of the system. These groups have to rely on written documentation of the system and guidance by the producer's staff. They "regularly encounter difficulties that upon consultation with the producer's installation and fault finding programmer are traced to an inadequate understanding of the existing documentation" (Naur 1992, p. 229).

Naur's conclusion to both stories is that with certain kinds of large programs, the continued adaption, modification, and correction of errors depends on certain kinds of knowledge possessed by a group of programmers who are closely and continuously connected with the software system (Naur 1992). For these systems the programmers' knowledge transcends that which is recorded in the documentation. Naur's argument is related to three issues: how, why and the application of how and why. A programmer possessing a theory of the program can explain *how* the software systems relates to the referent system. He is able to explain what real-life affairs the program is matched to. This involves everything from the software's overall characteristics, its architecture, to how its details are mapped into the program text and any additional documentation. The programmer is also able to justify *why* the program is the way it is. The justification is the programmers' direct, intuitive knowledge or estimate. It makes use of reasoning based on quantitative estimates, design rules, comparison with alternative.

Arguing for the third issue, the application of how and why, Naur says the programmer is able to constructively respond to demands of modifying the program. Based on the works of American psychologist Ryle, Naur calls this understanding a *theory*. In extension to this he argues that successfully developing software is a question of building this theory.

Writes Naur: “a person who has or possesses a theory . . . knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the activity of concern” (1992, p. 229). The notion of theory was proposed by Ryle (1949) in an effort to describe the difference between intellectual and intelligent behavior. Ryle claims that intelligent behavior is the ability to do certain things without having any concrete knowledge to build this behavior on. These are things we do without thinking about them, like talking grammatically correct, for instance. Intellectual behavior is, according to Ryle, a question of not only doing certain things intelligently, but also to explain and argue about the behavior. Behind intellectual behavior lies a theory. A theory is understood as the knowledge a person has to possess to act intellectually.

Intellectual behavior is not merely confined to the general and abstract part of insight. The person having the theory must have an understanding of the manner in which the central laws apply to certain aspects of reality, so as to be able to recognize and apply the theory to other similar aspects (Kuhn 1996). It is the knowledge of knowledge, so to say.

Relating this theoretical framework to his first case study, Naur argues that team B did not possessed the theory behind the compiler. Hence their suggestions to incorporate modifications not based on the program’s existing infrastructure. Team A possessed the theory behind the compiler, and were thereby able to classify team B’s suggestions as patching up and befuddling the simplicity of the original design. Team B did not have the theory to constructively respond to demands of modifications. Team A did. The same conclusion applies to his second case study.

Assuming a software development model is a set of work rules for programmers, saying something about what to do, in what order, which notations and programming languages to use and what kinds of documents to produce, argues Naur, then the most important aspect of the method is that of ordering actions. But: “In building the theory there can be no particular sequence of actions, for the reason that a theory held by a person has no inherent division into parts and no inherent ordering. Rather, the person possessing a theory will be able to produce presentations of various sorts on the basis of it, in response to question and demands” (Naur 1992, p. 236). Naur supports himself on the conclusions of more radical philosophers of science, that there is no such thing as a scientific method helpful to scientists (Feyerabend 1996). It is therefore a mistaken assumption, he argues, that software systems development should be based on scientific methods. “[T]he notion of methods as systems of rules that in an arbitrary context and mechanically will lead to good solutions is an illusion” (Naur 1992, p. 237), Naur argues referring to software engineering.

Discussion

Naur's perspective on software systems development is that of the individual developer. This is somewhat different to both software engineering and participatory design. Software engineering looks at the development effort with what is to be done when and by whom in mind—a bit crudely put, it could be said that software engineering is concerned with building the system right. Participatory design looks for ways to develop the right system, looking at how the software interacts in a larger social context. Naur does not concern himself with how software is to fit into the larger social context, but neither does he offer any advice on how developers best can build a theory of the software system being developed. His critique of software engineering as an illusion, and the claim that scientific methods are not much help to neither scientist nor software systems developer, makes Naur a representative of Hirscheim and Klein's subjectivist position.

While not the first to question the appropriateness of an approach to software systems development based on scientific methods, Naur is among the first to make the connection between software systems development and knowledge research. His contribution is significant in that it provides argumentation for viewing software systems development as a knowledge intensive activity. While implicit in some of the approaches described in the previous chapter, Naur is explicit on the fact the knowledge and understanding is the key factors in software systems development. He argues why it is so, but provides no practical clues as to how this would be reflected in an approach to software systems development.

Knowledge as a social activity

The way a task as it unfolds over time, looks different to someone working on it compared with the way the task looks when finished (Ryle 1954) (Bourdieu 1977). This is the foundations PARC Xerox scientists John Seely Brown and Paul Duguid base their research into understanding work practices. The two central terms here are are *opus operatum* and *modus operandi*. "The opus operatum, the finished view, tends to see the action in terms of the task alone and cannot see the way in which the process of doing the task is actually structured by the constantly changing conditions of work and the world" (Duguid and Brown 1991, p. 41). The Oxford Dictionary defines *modus operandi* as "an unvarying or habitual method of procedure" (Allen 1991, p. 763), and lists routine as synonym. This difference between actual practice and the abstracted description of it is central to Duguid and Brown's work on organizational learning.

Duguid and Brown's work is based on ethnographic studies of workplace practices, in particular Julian Orr's study of Xerox' field technicians'. Inspired by social anthropology (Geertz 1973) Orr says the best way to understanding what field technicians are, is by looking at what field technicians do (Orr 1996). Like the Scandinavian tradition of software systems development regard software as a means in the inherent tension between management and labor, Julian Orr studies the role work practices and knowledge of work practices play in the same context. Orr bases his findings on his ethnographic study of Xerox' field technicians. By looking at the gap between the espoused work practices laid down by management and the real work of field technicians, he shows the importance of noncanonical practices.

The two terms canonical and noncanonical practices are central in Orr's study. Canonical practices are formal descriptions of work. They are abstracted notions of how work, as espoused by management, is supposed to be. It is prescriptive. Canonic work practices can be understood as the *opus operatum*. It separates work from learning, and values abstract knowledge over actual practice. More significantly it separates learners from workers (Duguid and Brown 1991). For the field technicians canonical practices are laid down in directed documentation, which are stepwise guides for debugging and repairing copying machines.

From management's point of view, laying down canonic work practices is an effort to reduce the organization's vulnerability to staff turn over by codifying information previously in the domain of individual field technicians. By making the job less dependent on the individual, the organization is thought to be less vulnerable to flux in the work force. Seen from the point of view of the field technicians, canonic work practices is a tool in the hands of management to de-skill labor . More importantly, though, Xerox' field technicians also find the canonic work practices falling short of handling the complexity of everyday work.

Both directed documentation and the company's courses for training new field technicians are viewed by the field technicians as valueless. Both are aimed at finding a single point of failure through a decision tree. Where there is no single point of failure, or where the point of failure has yet to be documented, the directed documentation falls short. The gap between actual problems and the canonic practices are bridge by the field technicians' non-canonic practices, their *modus operandi*.

Narration is a central feature to the field technicians' work practice. The use of narration has two aspects. Narration is used by technicians working together to develop a causal map out of experience to replace the route directed by manuals and technical field guides provided by the company. It is used as a means of exploring possible causal reasons for a machine's breakdown, to keep track of sequences of behavior, and to relate personal hunches, insights, misconceptions and so on in an effort to dissect the problem and thereby increase their own understanding. Narration's second aspect is that of a collective reservoir of accumulated wisdom. Stories told are often incomplete and decoupled from context. They form background knowledge used in making meaning of an otherwise chaotic world.

Field service work is a situated practice, in which the context is part of the activity (Orr 1996). The field technician's work consists of building an understanding of a problem complex through talking with the customer and studying machine logs. This collaboration with the customer is important, but not the only significant communal activity undertaken to make meaning of the problem at hand. Orr also points to the importance of collective work in piecing together the information presented when a machine is faulty. Their work practice makes "do with whatever [is] at hand" (Lévy-Strauss 1966, p. 17). Their work practice isn't rigid and structured, but rather situated and thereby reflective of what problems the situation present them with. The field technicians form a community-of-practice, that "continue to develop rich, fluid, noncanonical view to bridge the gap between their organization's static canonical view and the challenge of changing practice" (Duguid and Brown 1991, p. 50).

Discussion

While the domain of Julian Orr's field technicians is distinctly different from software systems development, the central activity of repairing machines is not far removed from software systems development as Naur describes it. Both troubleshooting a copying machine and developing software is about building an understanding of the problem domain; building a theory to use Naur's words. Where Naur lacks any details about how to build a theory, Orr is directly concerned with this issue. Orr's conclusion is that making sense of the world, in other words building a theory, is a social activity that is situated within its context. It is an approach of trial and error, where making mistakes is equally important as being right. Each mistake reduces the problem domain's complexity, crystallizing the theory.

> The central issue in learning is becoming a practitioner not learning about practice. This approach draws attention away from abstract knowledge and cranial processes and situates it in the practices and communities in which knowledge takes on significance. Learning about new devices, such as the machines Orr's technicians worked with, is best understood (and best achieved) in the context of the community in which the devices are used and that community's particular interpretive conventions. (Duguid and Brown, p. 49)

The use of directed documentation and espoused work practices is relatable to software engineering's approach to software systems development. Both JSD and RUP lays out a map of the development effort through their steps and iterations. These stepwise guides are directed documentation as good as those used by Xerox' field technicians. It is fair to assume, underpinned by my personal experience with

RUP, that the guides suffer from the same short-comings as other types of directed documentation.

The mechanics of knowledge creation

In their book *The Knowledge Creating Company*, later summarized in the article *A Theory of the Firm's Knowledge-Creation Dynamics*, Japanese researchers Nonaka and Takeuchi lays down a model that describes the mechanics of knowledge creation. Knowledge creation can be understood in the context of developing an understanding of a problem as Orr shows in his study of field technicians. It can also be understood in the context of innovation, as in developing new ideas and new knowledge. Takeuchi and Nonaka—like Orr, Brown and Duguid— sees knowledge creation as a social activity. They go into great depths of explaining the mechanics of knowledge creation, expressing the social dimension by developing their own ontology of knowledge creating entities. Their epistemology, however, is based solely on the work of Michael Polanyi.

Before progressing with Takeuchi and Nonaka's theory of knowledge creation, it is worth dwelling briefly on Michael Polanyi for the sake of context. Polanyi was a Hungarian physician turned chemist turned philosopher of science. The critical component of Polanyi's work is an attack upon the ideal of objectivity as it was presented in science and philosophy at mid 20th century. His major contribution to the philosophy of science is the shift toward interest in scientific practice. With Polanyi shift of interest I come full circle with the research underpinning most of the material presented in this chapter. This shift is central to a great deal of later researchers both within the philosophy of science (Kuhn 1996). but also other scientific areas (Geertz 1973) (Bourdieu 1977) (Suchman 1987). It is reflected in the philosophers of science that Peter Naur leans upon when arguing that there is no appropriate scientific method for researchers (Feyerabend 1996). Polanyi's shift of focus from ideals to practice is also reflected in Duguid and Brown together with Julian Orr's research, when studying non-canonic work practices as opposed to canonic work practices.

Nonaka and Takeuchi, however, deal with the constructive philosophy of Polanyi's work. This work—as opposed to his critique of objectivity—represents Polanyi's developing interest in epistemology. Polanyi carefully works out his own epistemological model and sets forth a broad framework within which to think about knowledge as personal. His epistemology deals with two kinds of knowledge: tacit and explicit (Polanyi 1958). Explicit knowledge is formal and systematic. It is part of our everyday professional life, exemplified by manuals, books and lectures. It is the directed documentation we surround ourselves with in our professional life as software developers. It is the book used when learning a new programming language or a new development processes. These are hard and fast facts. It is quantifiable knowledge, but quantifiable knowledge is only the tip of the iceberg. There is more knowledge that can’t be expressed in words and numbers. Experience, subjective insights and intuitions is knowledge that can’t easily be expressed or shared, but nevertheless important knowledge. Polanyi calls this tacit knowledge. Tacit knowledge is something that is not easily visible and expressed. There are two dimensions to tacit knowledge. The technical dimension consists of skills and knowledge that is hard to pin down. This is the kind of knowledge and skills acquired over the years, but which are hard to explain or express. It can be exemplified by the expertise a master craftsman has acquired after years of experience. Tacit knowledge also has a cognitive dimension. This dimension consists of schemata, mental models, beliefs and perceptions so ingrained that we take them for granted. It is an image of reality, what is, and a vision for the future, what ought to be (Nonaka and Takeuchi 1998) (Polanyi 1958).

Building on Polanyi’s view of knowledge Nonaka and Takeuchi look at the mechanics and processes by which knowledge is created. The key to understanding knowledge creation, they argue, lies in understanding the creation of organizational knowledge. Knowledge creation takes place along two dimensions: epistemology and ontology. Their epistemology, their theory of knowledge, is Polanyi’s distinction between explicit and tacit knowledge. Equally important to their epistemology is the mobilization and conversion of tacit knowledge, their ontology. Ontology is concerned with the levels of knowledge-creating entities. To Takeuchi and Nonaka these levels are individual, group, organizational and inter-organizational. In this way they see knowledge creation, the construction of meaning, as a social activity more than simply an individual activity. Their view hinges on a “critical assumption that knowledge is created and expanded through social interaction between tacit knowledge and explicit knowledge” (Nonaka and Takeuchi 1998, p. 219). Knowledge is created and held by individuals. The challenge, as Nonaka and Takeuchi sees it, lies in sharing tacit knowledge between individuals throughout a group. They call this the knowledge spiral.

Socialization is the process of sharing experiences, and usually starts with building a field of interaction. This field facilitates the sharing of members’ experiences and mental models. It is exemplified by apprenticeship, internship, and on-the-job training where one learns through observation, imitation and practice. Socialization is situated within a concrete context, as mere transfer of context free and abstract information makes little sense to the recipient. Shared experience of the context is important for successful socialization.

Externalization is the process of articulating tacit knowledge into explicit knowledge. In this mode tacit knowledge takes shape as metaphors, analogies, concepts or models. The ambiguity of these images are considered important, as it encourages reflection and interaction between individuals. Through reflection, dialogue and interaction members articulate their hidden tacit knowledge that is otherwise hard to communicate. Conversion of knowledge this way is typically seen in the process of concept creation and is triggered by dialogue and collective reflection. This is not an analytical activity. It is the domain of analogies and metaphors.

Combination is a process of systematizing concepts into a knowledge system. Individuals exchange and combine and enrich different bodies of knowledge. It is a process of reconfiguring existing information through sorting, adding, combining and categorizing explicit knowledge. By networking newly created knowledge and existing knowledge form other sections of the organization, new products, services or managerial systems emerges. Nonaka and Takeuchi mentions an MBA eduction as another example of this type of knowledge conversion.

Learning by doing is closely related to the fourth mode of knowledge conversion: internalization. Internalization is the product of the three previous modes. Shared bodies of knowledge are internalize as shared mental models or technical know-how by the individual. This may happen without re-experiencing other people's experiences. Nonaka and Takeuchi writes:

> . . . if reading or listening to a success story makes members of the organization feel the realism and essence of the story, the experience that took place in the past may change into a tacit mental model. When such a mental model is shared by most members of the organization, tacit knowledge becomes part of the organizational culture. (Nonaka and Takeuchi 1998, p. 223)

Instead of describing a process model to facilitate the knowledge spiral, Nonaka and Takeuchi instead looks at the social factors that enables knowledge creation. They list five enabling conditions: intention, autonomy, fluctuation and creative chaos, redundancy, and finally requisite variety.

Intention is defined as "an organization's aspiration to its goals" (Nonaka and Takeuchi, p. 225). In a business environment this takes form of a strategy. For the intent and purposes of organizational knowledge creation, the essence of a strategy lies in the capability to acquire, create, accumulate and exploit knowledge. Organizational intention is also the most important criterion for judging the truthfulness of a given piece of knowledge.

Knowledge is created and held by individuals. Within the limitations given by the circumstances, individuals should be allowed autonomy. Autonomy is Nonaka and Takeuchi's second enabling condition. Autonomy is important to increase the chance of introducing unexpected opportunities. It also motivates the individual to create new knowledge. Original ideas emanate from autonomous individuals, diffuse within the team, and then become organizational ideas.

Fluctuation and creative chaos is the third organizational condition for promoting the knowledge spiral. Fluctuation is defined as "order without recursiveness"

(Nonaka and Takeuchi 1998, p. 228). During fluctuation members of an organization experience a breakdown of routines, habits or cognitive frameworks. They begin to question the validity of their basic attitudes toward the world. This breakdown fosters knowledge creation as new concepts have to be sought through dialogue. Chaos can be the result of an organization facing a real crisis (like sudden drop in productivity) or it can be provoked from within by leaders evoking a sense of crisis in the organization. As with fluctuation, chaos increases the organizations attention to define the problem and solve the crisis situation. In this way a crisis can considered be "creative chaos", where the problem to be solved is identified. This comes in sharp contrast with analytical information-processing where a problem is simply given and a solution is found through combining known information.

Chapter 4. Hacking

Both hacker and hacking, their usage, and their meaning are ambiguous. The two terms seem to have different meanings and different connotations to different people and at different times. As Gisle Hannemyr observes:

> The word *hacker* and *hacking* as applied to computer work is not very clear. Webster defines *to hack* as *to cut with repeated irregular or unskillful blows*, and *a hacker* as *one who forfeits individual freedom of action or professional integrity in exchange for wages or other assured reward*. These definitions, however, bear no resemblance to the common usage of the words *hacking* and *hacker* in the context of computer work. (Hannemyr 1998, p.255)

Little research literature exists on the way hackers develop software. It is therefore hard, unlike the preceding sections, to draw a picture of the state of affairs within the research tradition. There is, however, abundance of literature describing hacker communities. I will therefore use this chapter to bring the reader up to speed in the different ways hackers have been described.

Early hacker communities

The technical community surrounding MIT's Artificial Intelligence Laboratory during the 1960s and 70s were one of the first communities to explicitly call themselves hackers. Journalist Steven Levy (1984) describes this community in great detail, but not in a technical way. The book is more of an anthropological study of hacker communities. According to Levy, the hack is "a project undertaken or a product built not solely to fulfill some constructive goal, but with some wild pleasure taken in mere involvement" (Levy 1984, p.23). Hacking is not a well-planned project with a clearly defined goal as in terms of the end-result's usability. As Levy describes it, hacking is more like a gleeful technical experiment to explore the boundaries and capabilities of technical systems. Hacking is, however, not just a single-person enterprise.

Hacking is not just a single-person enterprise. Central to Levy's portrayal of the AI Lab hackers is the tape drawer. Keep in mind that this takes place in an era when software is still being saved on paper tapes. After having produced software through hacking, the hackers put all the tapes in the tape drawer for everyone else to use and refine the software. The tape drawer contained the tapes used in developing software: the compiler, the linker, the debugger, and other tools. New tools were added to the tape drawer, and existing tools were improved upon and sometimes replaced. While all the MIT hackers worked on their own projects, they cooperated on improving and maintaining the tools necessary for everyone to develop software.

Refining a tool developed by someone else and refine it would be perfectly in order as long as the original writer was given his due credits. As such everyone were free to improve and expand on the tools. Levy's hacking bears the hallmarks of experimentation and the pushing of technical boundaries, as well as a combination of individual and collective work.

Good tools were important to the AI Lab hackers. They focused on practice. Steven Levy calls it the *hands-on imperative*. Levy formulates it such: "Hackers believe that essential lessons can be learned about the system—about the world—from taking things apart, seeing how they work, and using this knowledge to create new and even more interesting things" (Levy 1984, p. 22). Important knowledge and understanding is accumulated through trial and error, through experimentation, not solely on reading and theorizing alone. To this end, anything that keeps the hacker away from improving the system, whatever system that is, is considered evil. This is the basis for the tenets that Levy's work has become most famous for—the Hacker Ethic. Gisle Hannemyr (1999) clarifies Levy's hacker ethic, formulating it into a set of imperatives that reflects the hacker mode of operations, followed by a set of position statements that reflects the hacker attitude. The imperatives are:

- reject hierarchies
- mistrust authority
- promote decentralization
- share information
- serve your community (i.e. the hacker community)

The position statements are:

- when creating computer artifacts, not only the observable results, but the craftsmanship in execution matters
- practice is superior to theory
- people should only be judged by merit (not by appearance, age, race or position)
- you can create art and beauty by the means of a computer

The AI Lab community was not the only hacker community to appear in the 1960s. Similar communities appeared in several other academic environments on both the east and west coast of the US. Especially strong were these communities at Stanford and Berkeley. Hacker communities have always been good a surrounding themselves with a shroud of mythology, usually created by themselves. During the years 1973-75 there was a cross-Arpanet collaboration to collect terminology from technical cultures across the USA. This effort came to be named the *Jargon File* (http://www.tuxedo.org/jargon/). It is considered the first intentional artifact of the hacker sub-culture, or *hackerdom* as it dubs itself. Since its inception the Jargon File has remained the central repository of hacker terminology, slang, and mythology. As such, it is a look at how the hackers view themselves. According to the Jargon File the hacker is a " ... person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary". According to the same source the hack is "... an incredibly good, and perhaps time-consuming piece of ... " design or programming. In this context hacking is the activity that leads to a hack.

Both Levy and the Jargon File can be said to romanticize hackers. One person who didn't see the romantic side of hacking is Joseph Weizenbaum. In his 1976 book, *Computer Power and Human Spirit*, Joseph Weizenbaum writes about the compulsive programmer:

> Whenever computer centers have become established, that is to say, in countless places in the United States, as well as in virtually all other industrial regions of the world, bright young men of disheveled appearance, often with sunken glowing eyes, can be seen sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to strike, at the buttons and keys on which their attention seems to be as riveted as a gambler's on the rolling dice. When not so transfixed, they often sit at tables strewn with computer printouts over which they pore like possessed students of a cabalistic text. They work until they nearly drop, twenty, thirty hours at a time. Their food, if they arrange it, is brought to them: coffee, Cokes, sandwiches. If possible, they sleep on cots near the computer. But only for a few hours—then back to the console or the printouts. Their rumpled clothes, their unwashed and unshaven faces, and their uncombed hair all testify that they are oblivious to their bodies and to the world in which they move. They exist, at least when so engaged, only through and for the computers. These are computer bums, compulsive programmers. (Weizenbaum 1976)

While Weizenbaum sees no harm in the compulsive programmer, he regards their culture as disappropriate and unhealthy. Their approach to software systems development is inappropriate. Weizenbaum does not dispute the fact that hackers are skilled technicians, but their way of software development is unstructured. The hacker has technique, but not knowledge. Using classic Baconian scientific theory to describe software systems development, Weizenbaum writes:

> He [the hacker] has nothing he can analyze or synthesize; in short, he has nothing to form theories about. His skill is therefore aimless, even disembodied. It is simply not connected with anything other than the instrument on which it may be exercised. His skill is that of a monastic copyist who, though illiterate, is a first rate calligrapher. (Weizenbaum 1976)

Looking at the title of his book, *Computer Power and Human Reason*, his views are understandable. Weizenbaum's agenda is to underpin software systems development as a purely rational endeavor. The hacker approach didn't fit with his view of rational behavior.

Towards the mid-1970s the AI Lab's hacker community began to crumble. Old hackers moved on to positions within the burgeoning computer industry. Steven Levy ends his book with a short epistle about "the last hacker", Richard Stallman. In 1984 Stallman looked as the last of a dying breed of hackers, but in reality he was the first of a new breed of hackers. More on that a bit later.

Unix

Bell Laboratories and General Electrics had been working on a joint venture since the mid 1960s. They were developing a multi user, time-sharing, multi-processor operating system based around a hierarchic file system. The operating system was meant to replace IBM's operating system, the Compatible Time-sharing System, or CTS for short. Bell and GE called their system Multics. By 1969 the Multics project was on the brink of collapse. Since 1968 the software developers by Bell Laboratories were becoming increasingly convinced Multics would never become the envisioned operating system. Under pretense of writing a word processor for AT&T Bell's patent office, Multics developers Ken Thompson, Dennis Ritchie, M.D. McIllroy and J.F. Ossanna set to developing their own timesharing operating system. This operating system was also to be based around a hierarchic file system.

During 1969 ideas were formulated, and using an old DEC PDP-7 that Ken Thompson had been implementing his own version of the game Space Travel, the file system was implemented. The filesystem would form the core of the operating system that would later be known as Unix. The name Unix is attributed another of Bell Lab's developers, Brian Kernighan, as a pun on Multics (think *uni* instead of *multi*). Every time Bell Labs got a new computer, Unix had to be reimplemented. The reason for this was that the entire operating system was written in assembler, code that is specific to a hardware architecture. Each reimplementation saw improvements in the operating system. By 1971 the word processor was ready for the patent office. In the process the Bell Lab developers had developed an operating system that remains popular till this day.

Probably the largest reimplementation of Unix took place in 1973 when Brian Kernighan and Dennis Ritchie created the C programming language for Unix systems programming. That year Unix was ported from assembly code to C. Unlike assembler, C abstracts hardware details. Instead of having to port the entire operating system every time a new hardware architecture is to be used, only the C compiler had to be ported. This gave Unix an advantage over other contemporary operating system, earning its name the portable operating system (Ritchie 1984) (Raymond 1999b).

1973 was also the year when Unix took its first steps into the world. That year Ken Thompson and Dennis Ritchie presented a paper on Unix. This took place at the

"Symposium for Operating Systems" held by Purdue University. Due to an anti-trust agreement between the federal authorities and AT&T Bell, the company was forbidden to market or sell anything but telecommunications equipment. They could not sell Unix for profit, so when professor Bob Fabry of Berkeley University asked Thompson and Ritchie if he could have a copy of their operating system, they were happy to oblige. January 1974 a tape containing Unix version 4 arrived at Berkeley. Shortly after the people at Berkeley were in desperate need of help. Unix crashed of inexplicable reasons. Via a modem line from the east coast Ken Thompson helped debug. This was the beginning of the cooperation between AT&T Bell and Berkeley University.

While AT&T Bell could not market Unix, people at Berkeley did. In 1977 Bill Joy created the first Berkeley Software Distribution, BSD for short. BSD consisted of the Unix operating system and a Pascal system Joy had co-developed with Ken Thompson when Thompson spent his sabbatical at Berkeley in 1976. By 1978 the BSD had already undergone dramatic changes, and Joy released a new version he called 2BSD. This new version included *termcap*, a universal screen driver, and *vi* a text editor Joy had written specifically for termcap. BSD continued to expand over the next years. The operating system was still AT&T's property, and all BSD users had to acquire a Unix license from AT&T.

While Unix was developed within a corporate atmosphere its development is central to the hacker culture. The original Unix developers are considered hackers, even though they work within corporations. This indicates that being a hacker is more a set of mind. Not only has Unix become the hacker operating system of preference, but the C programming language devised for Unix systems programming is the hacking culture's *lingua franca*. Eric Raymond says that C was designed to be pleasant, unconstraining, and flexible as an explanations of its popularity. He further says that both Unix and C were constructed from the "Keep It Simple Stupid" philosophy. Both these traits are, in his eyes, typical for the hacker mind set (Raymond 1999b. BSD's spread within the university system is probably one of the reasons for Unix' popularity, but a decision made by the Defense Advanced Research Projects Agency, an agency within the US Department of Defence, Department of Defence Research, was to further contribute to Unix' popularity.

The Arpanet

January 1986 the Internet Engineering Task Force (abbrev. IETF) started as a quarterly meeting of US government funded researchers. The IETF is responsible for developing and refining the basic technology of the Internet. The organization has never been incorporated as a legal entity, but has merely been an activity without legal substance. It is a membership organization without a defined membership. No criteria are required to partake in the organization's activities, but membership is only open to individuals not organizations or companies. Any individual who participates in an IETF mailing list or attends an IETF meeting can be said to be an IETF member. The IETF develops standards, not implementations, and it has set up a standardization process to support its activities (Bradner 1999).

Jane Abbate writes: “The ARPANET was born from an inspiration and a need” (Abbate 1999, p. 43). In 1960 Joseph C.R. Licklider had written a influential paper on human-machine interaction:

> The hope is that, in not too many years, human brains and computing machines will be coupled together very tightly, and that the resulting partnership will think as no human brain has ever thought and process data in a way not approached by the information-handling machines we know today ... Those years should be intellectually the most creative and exciting in the history of mankind. (Licklider 1960, pp. 4-5)

Licklider was director of the Information Processing Techniques Office (abbrev. IPTO) by the Advanced Research Projects Agency (abbrev. ARPA) until 1966. When Licklider resigned in 1966, Robert Taylor assumed the position. By now the IPTO was funding research in projects such as time-sharing, artificial intelligence and graphics in research centers across the United States. While each research center had its own sense of community, Taylor felt they were digitally isolated from each other. Where Licklider had had the vision, Taylor had the need. In 1967 the IPTO stated funding work on a computer network that would connect and link ARPA’s research centers and computing sites. The network was called the ARPA Net-work, or Arpanet for short.

> Roberts envisioned the ARPANET as a way to bring researchers together. He stressed early on that “a network would foster the ‘community’ use of computers.” “Cooperative programming,” he continued, “would be stimulated, and in particular fields or disciplines it will be possible to achieve ‘critical mass’ of talent by allowing geographically separated people to work effectively in interaction with a system.” (Abbate 1999, p. 46)

In 1968 the first four nodes of the Arpanet network was set up. It was hosted by educational institutions across the United States: the University of California at Santa Barbara, the Stanford Research Institute, the University of Utah, and the University of California at Los Angeles. Once the first four nodes was functioning smoothly, the IPTO expanded the Arpanet into fifteen nodes including MIT and the National Center of Super Computing Applications at Illinois. The fifteen node network was up an running in 1971. This connected the hackers at MIT’s AI Lab with similar communities at Stanford and Berkeley. This was the first on-line hacker community, and as mentioned earlier in this chapter, the Jargon File was a result of.

The Stanford Research Institute was given a contract to create an online resource called the Network Information Center (abbrev. NIC). The NIC would maintain a directory of the network personnel at each site within the Arpanet. They would create an online archive of documents relating to the network, and provide information on resources available through the network. An informal networking group, named the Network Working Group (abbrev. NWG), was set up to develop software specifications for the host computers within Arpanet. The NWG would provide a forum for discussing early experiences and experiments within the network. As the Arpanet was centered around educational institutions, the members of the NWG were mainly researchers and graduate students by these institutions. “The NWG would gradually develop a culture of its own and a style of approaching problems based on the need and interests of its members” (Abbate 1999, p. 59).

Through the 1970s the Arpanet grew to include several other new networks. By 1979 many of the original Arpanet hosts were reaching the end of their useful lifetime and had to be replaced. The heaviest cost of replacement was the porting of the research software to the new machines. ARPA had also experienced a difficulty in sharing their software because of the diversities of hardware and operating systems. They decided to standardize on one operating system. As choosing a single hardware vendor was found impractical due to the widely varying computing needs of the Arpanet sites, ARPA chose to standardize at the operating systems level. Unix was the chosen operating system, linking Unix even closer to the on-line hacking communities (McKusick 1999). In the fall of that year, Bob Fabry responded to ARPA’s interest in moving towards Unix by submitting a proposal for Berkeley to enhance 3BSD for use with the ARPA community. By the beginning of 1980 Fabry got a contract with ARPA to enhance BSD for the ARPA community, gaining wide distribution and visibility.

From the very start, the design decision behind the Arpanet had been to provide a means for any existing computer network to connect with the Arpanet. Contrary to contemporary competing technologies like the telecom industry’s X.25 initiative that required even local networks to migrate to the X.25 recommendation, the Arpanet provided a gateway to connect an existing network infrastructure with the Arpanet’s packet switched internet technology. As such the Arpanet was an internetwork, or an internet. Jane Abbate calls this the Arpanet’s “embrace and extend philosophy” (Abbate 1999).

By the early 1980s the Arpanet was still under military control. Even though several educational organizations were connected with the network, the majority of educational organizations were not. Unhappy with the situation people from within the Arpanet community and the US research institutions worked together to provide more widespread access to the network. In 1983 the US Department of Defense split the Arpanet in two: MILnet for military sites, and ARPAnet for civilian research sites. The split was made to separate the military’s operations from the research communities, and thereby allowing the research sites to manage the network according to their own needs. By the mid-1980s it was apparent that the old network infrastructure was becoming obsolete, and in 1987 the managers of ARPA's network program decided the network had to be retired. The entire network was moved from the obsolete networking backbone over to private, civilian networks that had been set up during the 1980s. This new network was called the Internet. The entire Arpanet community was moved over the Internet.

In 1984 leading contributors of the Network Working Group wrote a memo on Arpanet protocol policy, *ARPA-Internet Protocol Policy* (Postel and Reynolds 1984). This document is the basis for forming the IETF in 1986. The task force would focus on long-range technical planning. Their work process is based around working groups and documents called *Request For Comment*, or RFC for short. The process is described in more detail by both Jane Abbate (1999), Eric Monteiro (1998), and Scott McKusick (1999), but originally defined in RFC 1310 (Chapin 1993) and revised in RFC 1602 (Huitema and Gross 1994).

Working groups within the IETF coordinates their activities through e-mail. The task force itself holds meetings several times a year. The principle is that standards for the Internet are developed through consensus, after discussion among all interested parties and after proposed protocols have been tested in practice. That any one individua can participate in the IETF's activities, is to ensure that all interested parties really do get to express their view. Throughout the process the proposals are to be published electronically as Requests For Comments.

Is the IETF a hacker community? They don't view themselves accordingly, but their culture is very similar to technical hacker communities. They share information freely, and cooperate to develop joint infrastructure. I would claim, along others, that the IETF fits within an understanding of hacking as a collective way of developing and exploring technology. Its ties with other hacker communities are clear, with similarities from both the MIT hackers and the Berkeley Unix hacking communities.

GNU's not Unix

> Starting this Thanksgiving I am going to write a complete Unix-compatible software system called GNU (for Gnu's Not Unix), and give it away free to everyone who can use it. Contributions of time, money, programs and equipment are greatly needed. (Stallman 1983)

This is how Richard Stallman's initial statement for the GNU project starts. The statement was posted to Usenet on September 27 1983. Stallman had been part of the hacker community at MIT's AI lab since the early 1970s, and seen how the community dissolved as its members moved on to assume positions within the computer industry. At MIT Stallman had worked on their operating system, the Incompatible Timesharing System. He had worked in a community where software was shared, and was of the opinion that this sharing should be enforced outside MIT. Stallman's agenda is social. He is of the opinion that software cannot be owned by a single person or entity, but considers software a collective commodity that may be freely modified and shared by anyone.

Stallman's argument is based on the individual's freedom. By copyrighting software and providing it only as executables, individual freedom is compromised. Anyone should be free to read, fix, adapt and improve software, not just operate it. By taking away those rights, users loose freedom to control that part of their lives. Software should be free in the sense that it provides the right to read, fix, adapt and improve upon it. Said with Levy's words: "information wants to be free". Essential to this freedom is to always provide the software's source code for free. As long as the source code is available, free for anyone to do as they please with, freedom is achieved. Stallman call software that provide this freedom "free software". Providing the source code is not sufficient to protect the user's freedom.

To ensure that free software remains free, he created a the GNU General Public License (abbrev. GPL). The GPL is somewhat different from an ordinary end user license agreement. It starts with a large preamble stating the social intentions behind the license, what ends it is to serve. The license's main goal is to ensure that free software stays free even after modification. This is achieved by enforcing the GPL on any software using some or all of a GPLed software projects source code. This way the user's basic freedom is retained, and the amount of free software expands. Stallman also called this model *copyleft*. As an alternative to existing copyright laws that deprive users of their basic freedoms, copyleft "is a general method for making a program free software and requiring all modified and extended versions of the program to be free software as well" (Stallman 19??).

Emacs, the text editor Stallman had been working on since the mid-1970s, was the first program to become a part of GNU. A compiler and debugger for C was next. Stallman, who Steven Levy in his after-word to *Hackers* portrayed as the last of a dying breed of true hackers, gained increased support for the GNU project. Remaining true to his ideals, Stallman has worked on GNU and software freedom since the initial announcement in 1983, today being the chairman of the Free Software Foundation, a non-profit organization that works for free software.

The Internet hackers of the 1990s

By the late 1980s and early 1990s GNU had become an extensive collection of tools. Upon having installed the basic Unix operating system, most systems administrators would download and compile the GNU tools for their computers.

While the GNU tool chest had grown large and extensive, Stallman still lacked an operating system kernel. This was to be soon remedied by a young Finish computer student.

> Do you pine for the nice days of minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a OS you can try to modify for your needs? Are you finding it frustrating when everything works on minix? No more all-nighters to get a nifty program working? Then this post might be just for you :-)
>
> As I mentioned a month(?) ago, I'm working on a free version of a minix-lookalike for AT-386 computers. It has finally reached the stage where it's even usable (though may not be depending on what you want), and I am willing to put out the sources for wider distribution. It is just version 0.02 (+1 (very small) patch already), but I've successfully run bash/gcc/gnu-make/gnu-sed/compress etc under it. (Torvalds 1992)

Like Stallman eight years earlier, Linus Torvalds posted his initial announcement on Usenet. Torvalds was interested in a Unix system to run on his low-end home computer. As a student could not afford the expensive hardware and almost equally expensive license for commercial Unix systems. Minix, a Unix-like operating system written by Andrew Tannenbaum, was freely available at the time. Hurd, the GNU operating system kernel, seemed to be nearing completion. Still, for the fun of it, Torvalds chose to write his own operating system kernel. "This is a program for hackers by a hacker," as Torvalds puts it in his initial announcement Bundling the Linux kernel with the GNU tools, the GNU/Linux operating system rapidly attained interest within the on-line hacker community.

By the early 1990s the ARPAnet had become the Internet, and was widely available with educational institutions across the United States and Western Europe. The Internet had become an important communications channel for the hacker communities. Usenet provided services for on-line discussion, and files were shared using FTP. SunSITE had become a central repository for freely shared software. Linux was freely available from an FTP site in Finland, and developers across the Internet started contributing code to the kernel. After a while Torvalds decided to provide the Linux kernel under the GPL.

Linux, however, was not the only project with contributors across the Internet joining to write software together. The widely popular programming language Perl is developed this way, and has been since 1988. Most of the freely available software found at SunSITE never saw the same attention as Perl or Linux. Most were developed by a single individual or a small group. Some of these groups were geographically co-located, others were distributed across the Internet. It was this Internet based distributed software development that caught long-time hacker Eric Raymond's attention

In 1998 Raymond wrote what has become a seminal essay on Internet based distributed software systems development, *The Cathedral and the Bazaar* . With background in his own, fairly popular software fetchmail, he wrote about the new wave of hackers. Raymond sees the Internet based distributed development approach as a way of making “software that doesn’t suck”, considering this way of working as an effective means of parallelizing the debugging effort. Like Stallman, Raymond is of the opinion the a program’s source code should be open and freely available. Raymond, unlike Stallman, does not consider the social side of freely available source code. On the contrary, he sees the use of other people’s source code as a way of “scratching his own itch”, another of his mottoes. This kind of hacking is just a convenient way for several people to pitch their effort together to write high-quality software. To indicate his differing opinion, Raymond labeled his view of software as “open source”.

Particular to this way of developing software, open source software development projects don’t put any limitations on the communication between participants. These projects often use a Usenet group or a mailing list for communication, free and open for anyone to participate. Raymond calls it the bazaar, as opposed to the cathedral way of developing commercial software where everyone have to follow a creed or methodology.

Looking at *The Cathedral and the Bazaar* there are elements reminding of of participatory design in hacking, says Hannemyr (1998). There are elements of rapid prototyping, iterative development and strong user participation. But there are differences as well. Hacking lacks the formalism found in more traditional development methodology. There is no clear definition of roles, and the careful planning of iterations is non-existent. Raymond does not take his users’ participation for granted. Instead he lists three necessary pre-conditions for his work practice to be applicable: 1) The projected system must fill an unfilled personal need for the instigator; 2) the project needs to secure user participation and maintain continued user support; and, 3) the instigator must have good interpersonal and communication skills.

Hacking in perspective

Gisle Hannemyr is correct in that the word hacking is ambiguous in its use and meaning. I have presented several communities that are considered hacker communities, even though not all call themselves hackers. Gisle Hannemyr sees hacking in the context of software systems development. He bases his observations on personal experience and to a great deal on Eric Raymond’s *The Cathedral and the Bazaar* . He writes:

> When asked: “In your work, do you view yourself as a ‘tinkerer’ or an ‘engineer’?” one hacker answered: “Any real developer has to be both. This is what you have to learn

> from transmission outside the scriptures, from working with other people: When you have to be bottom-up and when you have to be top-down.” This brief statement summarizes what I consider the three most pronounced aspects of hacking: The emphasis on skills acquired through practice (“outside the scriptures”), the importance of the community (“working with other people”), and the equal emphasis put on engineering (“top-down”) and bricolage (“bottom-up”). (Hannemyr 1998, p. 258)

Hannemyr borrows Lévy-Strauss’ term *bricoleur* to describe hacking. The bricolage is a useful way to think of piecing together. The point of bricolage is the reflective use of a closed set of resources—typically what is at hand like things, understandings, facts—to accomplish a defined goal. The set is the accumulation of previous manipulations, one’s experience and knowledge, and, in literal bricolage, physical objects. Orr (1996) shows the bricolage as an apt description of how technical work gets done. For the hacker, both Levy (1984) and Raymond (1998) shows that the hack, which is the goal, can be something else than a piece of software as long as it a technical achievement.

Hannemyr draws up the defining dichotomy of hacking: engineering vs. bricolage. In this dichotomy engineering is understood as a analytic and systematic approach to software systems development, while the bricolage consists of applying a given item from the closed set of resources toward the goal of the bricolage, possibly by some reinterpretation or modification. As hacking has elements of both, practice and experience is the key to understanding which approach works better in a given situation. The hackers’ approach to software systems development isn’t purely analytical. Instead it contains elements of playfulness and experimentation.

Hannemyr also notes the elements of participatory design in Raymond’s works, but sees hacking as a counter-movement to scientific management. Hannemyr describes the situation as follows:

> The most profound effect of application of Taylorist principles to computer work was the introduction of a detailed division of labor in the field. Computer workers found themselves stratified into a strict hierarchy where a “system analyst” was to head software development team consisting, in decreasing order of status and seniority, “programmers”, “coders”, “testers” and “maintainers”. Then, below these on the ladder was a number of new adjunct positions created to serve the software development team: “computer console operators”, “computer room technicians”, “key punch operators”, “tape jockeys” and “stock room attendants”. Putting the different grade of workers in different locations further enforced the division of labor. Most corporations in the sixties and seventies hid their mainframes in locked computer rooms, to which programmers had no access. This isolated programmers from technicians, diminishing their social interaction and cutting off the opportunity for the exchange of ideas. It also prevented programmers from learning very much about the workings of the machine they programmed. (Hannemyr 1999)

Building on Lévy-Strauss' bricolage, Dahlbom and Mathiasen (1998) coined the two terms tinker and engineer. They argue that illiterate thinkers have bricoleurs and tinkers, while modern societies have engineers. The engineer is top-down while the tinker bottom-up. However, Hannemyr argues, even though hacking has elements of bottom-up working, they are very much engineers. In his eyes, de-skilling and the division of labor deprives software systems development of the engineering elements and reduces it to tinkering. Hackers see this, and their abhorrence of scientific management and tayloristic principles are based on their belief that software systems development is both an art and a craft. The aspect of art and craft is the crux of Hannemyr's description of hackers and hacking .

Tove Håpnes studied the hacker sub-culture by the Norwegian Institute of Technology. She viewed the sub-culture from a sociological point of view. Unlike Weizenbaum's compulsive programmers and Levi's techno-wiz hackers, she saw the culture as an ambiguous project. It is torn between both the individualistic and collective (Håpnes 1996).

> We find elements of *competition*, but also of *collaboration*; *play* and *entertainment* as well as *work* and *utility*. Hackers *consume* technology, and they *design* technology. They talk about *winning* and *mastery*, but also the importance of being *artistic* and *interactive*. The computers are *instrumental* as well as *expressive*; defined as "objective" tools which also include certain values, and subjective elements are integrated in their use. (Håpnes 1996, p. 145)

All in all, concludes Håpnes, the "… hacker culture is a form of *socio-technical experimentation* … Their [the hackers'] motivation is simply to challenge the boundaries of computers and their own creativity" (Håpnes 1996, p. 148).

It is obvious that hacking involves quite a special kind of relationship between individual and collective work, but in what way does this influence on the software systems development and what is this relationship in practice? There also has to be more to hacking than just this special kind of relationship. Hacking has produced numerous important innovations, like ground-breaking operating systems and the world's largest computer network. The picture drawn from existing sources is an unclear one, and the question still remains: what is hacking really, and how does it relate to software systems development?

Chapter 5. The Apache Web server

I have based this thesis on a case study of the group developing the Apache Web server. Before looking closer at the case study itself, I will give a quick overview of the project's background and the technology they are working on.

An introduction to the World Wide Web

While being employed by the international CERN laboratory in Switzerland, Tim Berners-Lee had been working on a documentation system for better organizing the research material produced on the site. The basis for his efforts was the idea that it would be better to organize scientific material in a way that resembles the human mind—as semantic knowledge networks. The basis technology for his project was hypertext. By making use of hyper textual links, Berners-Lee found it easier to connect relevant research material together. As he simply wanted to retrieve documents from and put documents to servers, he opted for a simple network communication protocol for this new technology which he had given the name the World Wide Web (Berners-Lee 1999).

The technology

The network communications protocol, Berners-Lee called the *Hypertext Transfer Protocol*, HTTP for short. It is a stateless network protocol piggy-backing on TCP/IP's application layer. At heart the Web is a traditional server/client architecture. Through a HTTP client software, the user requests documents on a HTTP server. The server responds by returning the requested document, and then close the connection. A client in this context is "[a]n application program that establishes connections for the purpose of sending requests" (Berners-Lee and Fielding 1996, p. 4). For the end user, the HTTP client is a Web browser, but it can in reality be any piece of software using HTTP to initiate communicate with another HTTP enabled piece of software. The server is "[a]n application program that accepts connections in order to service requests by sending back responses" (Berners-Lee and Fielding 1996, p. 4). Apart from the data transfer protocol, there were two more crucial elements of the Web: the Universal Resource Locator and the Hypertext Markup Language.

The Universal Resource Locator, URL, is the common naming format devised by Berners-Lee (Berners-Lee et al. 1994). It is meant as a meta naming scheme for the World Wide Web to identify both host and service. The URL consists of three elements <`the network protocol`>://<`server name`>/<`file path`>. During HTTP transactions the protocol element is http. While the URL was conceived for the Web, Berners-Lee realized that users need not limit themselves to just one network protocol. Instead of ignoring the existing Internet protocols and standards, he chose to embrace them with his own technology. That way it is possible to retrieve documents from, say, a FTP server using a Web browser. The URL's second element, the server name, is an IP address or host name. The URL's final part, the file path, indicates what file to retrieve from the server. File paths are handled in a traditional manner with directory structures and file names.

The third and final piece of Berners-Lee's Web technology is his document markup language, the Hypertext Markup Language, or HTML for short (Berners-Lee and Conolly 1995). While it is possible to transfer all kinds of files and documents using HTTP, Berners-Lee wrote his own documentation language to support the linking of information on the Web. In the spirit of his embrace and extend ideology, Berners-Lee decided to base it on SGML as that was one of the major document formats used at the CERN research site. With this markup language scientist would be able to embed the hypertext links that Berners-Lee considered the core of his technology. However, also part of his embrace and extend strategy, it would be possible to create links to document in non-HTML formats.

Growth

With the three basic building blocks in place—the URL, the HTTP, and the HTML—Berners-Lee's first steps in spreading the his new technology was to show its practical use. After much work he convinced a local systems operator at CERN to use the technology for the research site's on-line address book. Berners-Lee wrote the software himself. He had first written the HTTP client and server for his own Next computer. Next was even then an obscure operating platform, and in order for the software to run at CERN it would have to be ported to Unix. Being connected to the Internet through CERN, Berners-Lee tried his luck recruiting someone to help do the port. The effort was fruitless, and he ended up porting the server himself. The port would come to be known as the CERN httpd. The *d* at the end of httpd indicates that the Web server runs as a daemon process. Daemon processes are Unix processes that that runs in the background.

Berners-Lee did not copyright his software, he did not charge for its use, but merely requested to be attributed for his work on the technology. He then posted the source code to several USENET groups. The immediate response was minor, but in time there grew a substantial community around the Web technology. With time the community and user base grew so large that Berners-Lee realized that he alone could no longer direct and enhance the Web. He realized that he needed a way to direct and unify the movement surrounding the Web. As Berners-Lee did not want to commercialize the technology, he first tried pitching it to the IETF with no success. His first Internet proposal, the URL, took too long to establish as a factor.

A bit disillusioned Berners-Lee started looking for other approaches. He made contact with the people who had set up and managed the X Consortium. The X Consortium is an open industry consortium that develops and maintains the X Windows graphical user interface. With their help Berners-Lee founded the World Wide Web Consortium, the W3C for short, that was to handle future development of the Web. The W3C is today an industry-wide consortium with participants from major computer companies, educational institutions, governments, and more. It is the maintainer of the Web technology developing new standards for the Web community. Yet, the networking part of the Web— the URL and the HTTP—is handled by the the IETF.

A patchy Web server

Several individuals got into Berners-Lee's technology from the start. Unfortunately for Berners-Lee hopes of spreading the Web, most of these individuals were interested in his technology as a curiosity, something to implement for their own needs. They did not share Berners-Lee's visions of a knowledge network. The early Web browsers were either implemented in queer programming languages making them impossible to port, or they were dropped like a hot potato once the term project was over. Always on the cutting edge of Internet technology, one group that got into the Web early on was some computer enthusiasts at the University of Illinois National Super Computing Applications center. Of these Rob McCool and Marc Andreesen were the most prominent. They were developing their own Web browser called Mosaic, and their own Web server, the NCSA httpd. Both applications extended and improved Berners-Lee's original Web browser and server. Probably the most visible new feature added by the NCSA group was their browser's multimedia capability. Their Mosaic browser was the first to incorporate both pictures and text in a single web page.

The original NCSA Web server can be said to be a product of the experimental approach attributed hackers and hackerdom. It was a relatively good piece of technology, but far from bug free nor a particularly neat piece of programming. Its source code had been released along with a non-restrictive license, and the software was being used by the increasing number of commercial and educational Web servers across the United States and Europe. Because of its many bugs and somewhat lacking functionality, there grew a community around the NCSA Web server's code base. It was a community consisting mainly of Web masters—systems operators running their own Web sites—who shared in their individual work to make the NCSA Web server more bug free. They were collaborating to enhance their joint technology, sharing *patches* that fixed bugs in the original code base and enhanced it with new features.

"A patch is [a] temporary addition to a piece of code, usually as a quick-and-dirty remedy to an existing bug or misfeature. A patch may or may not work, and may or may not eventually be incorporated permanently into the program" (Raymond 1998, p. 349). The issue at stake was that neither of these patches had been integrated with the Web server's code base. This raised two problems. The first was one of installation. Installing the NCSA Web server had become quite a task. First the original source code had to be downloaded. Then it was a question of locating, downloading, and applying the important patches in circulation before compiling source code. At this point the second problem surfaced. As there were a fair amount of patches in circulation, the would-be system operator of an NCSA Web server could never be certain that none of the patches applied were in conflict with each other. Such a conflict would at best be time-consuming to track down, at worst next to impossible as the would-be Web master had little or no knowledge of the internal workings of the NCSA Web server.

This loosely organized group of Web masters were largely ignored by the NCSA development team. By early 1995 the current NCSA Web server release `1.3` had been out for the better part of a year without its original developers having shown any interest in updating the source code. None of the many patches circulating on the Internet had been applied to the original code base. Disillusioned by the NCSA developers' lack of interest in the Web server, this group came to organize itself around a mailing list they called new-httpd. It was a forum where they could meet and exchange their patches, cooperating on making it easier to install the Web server.

The original NCSA software was beginning to look more and more like a patchwork quilt. The new-httpd crowd were beginning to realize that the original software needed an overhaul. With the existing licensing policy of the NCSA server it was possible for the new-httpd crowd to use the original NCSA Web server code in their own product without problems. While the initial intention was to form a community that shared their patches, eliminating redundant work where several people fixed the same bugs, they soon came to realize that they wanted to create their own HTTP server. By February 1995 the patches were piling up. Something had to be done. The new-httpd crowd chose to unleash their wry tongues on the original NCSA code base. Like Unix had once been a pun upon its predecessor Multics, the new-httpd crowd chose to call their Web server Apache, a pun based on the fact that the NCSA Web server had become "a patchy" Web server.

The case study

The empirical basis of this thesis is a case study of the Apache project. I am looking at the project from its inception in early March 1995 until the release of Apache `1.0` by Christmas the same year. As mentioned above, the project's inception comes as a response to the state of the NCSA web server. Upon entering the story the Apache develops do not see themselves as more than a group of disgruntled users taking matters in their own hands. In fact, one participant says that the biggest problem facing the group is the newly released beta version of the NCSA `1.4` web server. Will the Apache group have the right for life, a justification for their dissident activities, if the NCSA group finally gears up and released a new version of their software?

The Apache project has no formal organization. It is not a membership organization. No prior requirements have to be met to participate. Anyone wanting to participate may do so, although most participants do contribute code to the project. There are formally no leaders. Every participant has an equal say in the decision making process. The project is made up of people using the NCSA web server, but looking to improve the software. Some of them have been exchanging bug fixes and feature enhancements for a good while. All participants are volunteers, developing Apache on their spare time. Almost exclusively all participants have full-time jobs or attend studies for a higher eduction. They are all male, they all live in the United States or Western Europe, and they all have various kinds of higher education. While the demography of the Apache team is not a part of this thesis, I will try to give an impression of the demography by short presentations of the most prominent members during the course of the next chapter.

It is difficult to know what to call the group of people hanging around the new-httpd mailing list at this stage. They are undeniably a community. As such I could have called them the new-httpd crowd. But the mailing list's name, new-httpd, suggests they are doing more than just improving the existing NCSA server. So they are more than a just a community. They are a community set to developing a new Web server, a community of developers working on the Apache web server. I have therefore chosen to call the community the Apache group, and what they do the Apache project.

Members of the Apache project are geographically distributed. During this early stage of the project the majority of participants are found in North America, but there are a few participants can be found in Western Europe. The tools used in coordinating the development effort are crude compared to the many group ware offerings available on the market today. E-mail and FTP are the primary tools used in coordinating the team. This is sufficient for their needs. To coordinate the effort the developers need means of communication, e-mail, and file sharing, FTP. The mailing list is hosted on Brian Behlendorf's Internet host `hyperreal.com`. `hyperreal.com` forms the group's focal point as it also host's the project's FTP archives and their Web pages. The FTP server is assigned an Incoming directory, where contributors upload their patches.

While the Apache project has no formal organization, it does have an institutionalized development process. The process is based on peer review, where all new code submitted has to be voted upon before it can be integrated with the with the code base. New code is submitted in the form of patches. Votes are based on the participants' experiences through applying patches to their own web servers. Once enough patches have been tested, a round of voting is announced on the new-httpd mailing list. The announcement includes the list of patches up for voting. Everyone can cast their votes. Each developers casts one vote for every patch on the list. Voting is handled through a numeric system. A vote of +1 for patches that should be included with the next release, a 0 to signal indifference about the inclusion or exclusion of a patch, and -1 to veto the patch For a patch to be integrated with the official Apache code base, it needs at least +3 vote points and no vetoes.

In this patch and vote system there are two semi-formal roles, the *version builder* and the *vote coordinator*. While the *Apache voting rules and guidelines* implies that the voting coordinator role need not be assigned a specific person who holds the title over time:

> A voting session can be initiated by anyone so long as a volunteer or volunteers can be found to:
>
> - be the vote coordinator: collect the votes cast by group members

in reality one person holds this title over time and hands it off to his successor (Thau, posted on new-httpd mailing list August 31 1995). The vote coordinator's task is to initiate the call for votes on new patches submitted since the last release. Upon initiation the round of voting is to be given a deadline, at which the voting ceases and the poll is handed over to the version builder.

The *version builder* is to "apply approved patches to create the *NEXT* version of the system" (Hartill and Fielding 1995). The version builder is also supposed to be "making the changes called for by other approved (non-patch) action items, adding the approved action item descriptions to the change log, and incrementing the version number" (Hartill and Fielding 1995). Upon doing this, the source code is rolled into a tar ball and uploaded to the Apache group's FTP site.

To keep track of their releases, the Apache team makes use of a version number scheme. The scheme consists of two mandatory elements, a major version number and a minor version number. The version number is on the format `<major>.<minor>`. The major version number indicates the software's generation. Throughout most of the period I look at in this thesis, the major version number is 0. Once the Apache group views their code mature, they increment the major version to 1. The minor version number is used to by the version builder to indicate that he has released a new version of the software with new functionality added to it. The minor version number is incremented by one when rolling a new tar ball.

There are also one optional element used in the version scheme, the branching revision. The branching revision is an update to a minor release. It is branching in the sense that the release is meant to address bugs specific to a previous release.

Instead of requiring a major upgrade of the software, the branching release only fixes bugs existing in a previous release. The branching release is therefore always exclusively a release containing bug fixes and no functionality. The branching release number is appended with to the minor version number using a dot as separator, resulting in something like `0.6.1`, where the final digit, one, is the branching release to minor release `0.6`. The branching release numbering scheme follows mostly the same rules as minor version, but the release manager does in periods deviate from the scheme to add extra information to the release.

Since February 1995 every e-mail sent to the new-httpd mailing list has been archived. It is reading these archives that form the empiric basis of this thesis. Before presenting excerpts of this data, I will go into some detail of the research method I have employed.

Chapter 6. Method

Before progressing to the case study, I will clarify my methodological approach. This chapter consists of three parts. The first part is a general discussion about the two major research traditions within computer sciences. I draw up a dichotomy between the positivist and interpretivist approaches. By highlighting the strengths and weaknesses of each approach, I use that as a basis for my choice of methodological approach. The second part is a description of the practicalities of the research work undertaken for this thesis. The final part is a discussion about the implications of my methodological approach on the case study and end-results derived through an analysis.

Research traditions within computer science

Computer science has its foundations in the empiric sciences. This positivist approach to science is characterized by repeatability, reductionism and refutability (Galliers 1992). It excludes from research everything but the natural phenomena or properties of knowable things, together with their invariable relations of coexistence and succession, as occurring in time and space. It assumes the observations of the phenomenon under investigation can be made objectively and rigorously. Using Hirscheim and Klein's (1989) taxonomy, this scientific approach lands squarely within the objectivist dimension. The researcher is considered a neutral party whose role is observation and recording. Scientific knowledge in this paradigm, is proven knowledge. Scientific theories are derived in some rigorous way from the facts of experience as recorded by the scientist. Science is based on what the researcher can see, hear and touch, etc. Personal opinion or preferences and speculative imaginings have no place in science. Science is objective, and scientific knowledge is reliable because it is objectively proven knowledge. In positivist science, "[s]cience is a structure built upon facts" (Davies 1968, p. 8).

The advantage of positivist research is that it can identify the precise relationships between chosen variables. Using analytical techniques the aim is to make generalizable statements applicable to real-life situations (Chalmers 1978). Through controlling the number of variables, complexity is reduced. Reduced complexity generates less noise, allowing for a closer study of the variables (Galliers 1992).

The approach of limiting variables has proven unfruitful within several strands of computer sciences. For traditional artificial intelligence it has lead to a dead end. By limiting the variables, the field has been able to find solutions to toy problems, but fails in generalizing the research to make it scale to real-life problems (Dreyfus 1999). There is also a problem applying the positivist approach to social situations:

> The empiric-analytical model [read: the positivist approach] is the only valid approach to improve human knowledge. What can't be investigated using this approach, can't be investigated scientifically. (Galliers 1992, p. 148)

As repeatability of a phenomenon is a prerequisite for positivist research, it is practically impossible to apply this approach to studying phenomena lacking repeatability, like social situations for instance. Alternative approaches must be sought.

Interpretative research assumes our knowledge of reality can be achieved through social constructions such as language, consciousness, shared meanings, documents, tools, and other artifacts. This kind of research tries not to predefined dependent and independent variables, but focuses on the complexity of human sense making as the situation emerges. It is an effort to understand the context of the phenomenon under study, and the process in which the phenomenon influences the context and the context influences the phenomenon (Klein and Meyers 1999).

Interpretivist researchers argue that the positivist approach is inappropriate in social scientific research because "interpretivist researchers do not subscribe to the idea that pre-determined set of criteria can be applied in a mechanistic way" (Meyers 1999, p. 68). This is because there is a possibility of many different interpretations of social phenomena. It is also argued that the social scientist himself has an impact on the social system being studied. The prime concern however, is that positivist researchers believe in forecasting future events concerned with human activity based on the assumption that patterns observed in the past will repeat themselves (Galliers 1992). In contrast, interpretive researchers insists that no such *a priori* fixed relationships within phenomena exist, but rather that observational organizational patterns are constantly changing (Klein and Meyers 1999).

The strength of interpretative research lies in presenting reality as an in-depth, self-validating process in which presuppositions are continually questioned. Through this process our understanding of the phenomenon is refined. The weakness is that its interpretative nature relies on the researcher's ability to identify his biases and assumptions. The danger is that even though identified, the researcher's biases still clouds the interpretation of the subject. This, however, is not simply a danger of interpretative research. Positivist research has even shown not to be free of preconceptions that cloud the results (Klein and Meyers 1999).

I am choosing the interpretative approach. I do this because I want to uncover the implicit arguments and assumptions that form the basis of the Apache group's approach to software systems development. The nature of studying a case through reading its mailing list archives preclude an empiric-analytical approach. The situations I am observing can't be repeated.

The research process

The interpretative approach to computer sciences is an iteration between fragments and the context (Klein and Meyers 1999). This has been the main approach taken throughout working on this thesis. In this part I retrace the steps taken in collecting and interpreting data. Looking back at the work, I was working within two hermeneutic circles: the first circle being to understand what would be required of a case study, the second circle the Apache project itself.

Choosing a case study

I spent the first two months of working on this thesis by looking at appropriate candidates for case study. First I identified a large number of interesting projects that could be potential case studies. My primary concern during this phase was that I wanted to study hacking, but I had no clear idea of how best to approach the matter. What I basically did, was that I surfed the World Wide Web looking for projects connected with the Linux operating system and the IETF standardization body. After a scant month of work, I had a handful of potential candidates. Among these were the Linux kernel, the Debian GNU/Linux distribution, the IPv6 and LDAP IETF working groups, the KDE desktop environment, the PHP programming language, the Perl programming language, and the Apache Web server just to mention those that were my favorites at the time.

Once a handful of potential candidates had been identified, I started looking for an approach to select the most appropriate case study among these. The approach chosen was to identify criteria a case study needed to fulfill for it to be approachable. This was an iterative process. For each iteration I identified some new criteria, and for each iteration candidate after candidate was excluded. Each time new criteria were formulated, I had to look closer into the remaining candidates then compare my findings in order to find new criteria for my case study. With each iteration I delved deeper into the remaining candidates. In the end I was left with three potential candidates that seemed to fulfill the criteria identified. These were the Linux kernel, the KDE desktop environment, and the Apache Web server.

The criteria

The criteria used in selecting the right case to study can be split in two. There are the criteria directly connected to what I wanted to study. They stem from my initial concern, that I wanted to study hacking in practice. The other type of criteria are the pragmatic kind. These criteria are connected with how good the data foundation for a case study.

The aim of this thesis is to study the process of software systems development within a hacker community. For the best possible historical documentation, I chose to look into a distributed hacker community whose primary mode of communication is e-mail via a mailing list. That way, if the community keeps an archive of its mailing list, I would have close to complete readouts on the development process. The first criteria for choosing a case study was therefore for it to be a hacker community. The second criteria consequently became that the community had to be geographically distributed, and thereby depend on e-mail communication. Not just any distributed team would do, for there had to be an archive keeping complete records of the community's activities. A complete mailing list archive was therefore the third criteria. A fourth pragmatic criteria was added, that the e-mail archives had to be more or less complete, not fragmented, as important bits could not be missing. These four criteria were connected with the traceability of the development effort, requirements that the case had to fulfill in order to form a sufficient data basis for a case study.

To be able to study the development effort over time, it became apparent that the mailing list archives had to be of a certain size, that it stretched itself over a period of time. This would be the only way to study the development team and its practices evolved over time. No fixed time-frame was set for this eighth criteria, but it was implicit that the mailing list archives needed to stretch over several years in order to be of any use.

Next I looked closer at the kind of project I wanted to study. Small-scale projects do not suffer under the same inherent problems as larger, more complex software systems development efforts. This lead to a fifth criteria for choosing a case study, that the software development effort undertaken by the hacker community had to be significant. An absolute size was not set, but it was implicit that the project needed to be larger than one single individual could possibly keep track of himself. I was looking for a case where total opacity and top-down control was simply not possible due to the sheer size of the project. There was another side to the matter of the fifth criterion. To better reflect the reality of the software industry we were not interested in studying a group of close friends knowing each other well, but rather a large community with the occasional conflict and clash of personalities. This was the sixth criteria.

As I also wanted to look at innovation in software systems development, potential candidates had to be more than simply *chasing tail-lights*—as Valloppolli (1998) has observed about a number of hacker projects—the mere imitation of previous work. The development effort had to push the boundaries of software in a way that could be studied. From this, the sixth criteria gave itself, that the software development effort had to be innovative and create new technology. From this followed a seventh criterion, more of the pragmatic kind, that the technology of the development effort had have a broader appeal. It had to be more than an obscure encryption algorithm, no matter how innovative it might be. The technology had to have a broader appeal outside of the community. Said another way, the case study had to be of such a kind that other people would be interested in reading about it.

The final criteria never gave itself until I had chosen a case study that had to be rejected. The ninth and final criteria: that important decisions and arguments had to take place on the list. My first case did not reflect the executive decisions, nor how they were being made, and as such proved impossible to pursue. The nine criteria for choosing a case study consequently became:

1. The case has to be a hacker community.
2. The community has to be geographically distributed.
3. The community's prime channel of communication is e-mail via mailing list(s), for which there are archives.
4. The community's e-mail archives have to be more or less complete. Important or large sections should not be missing.
5. The mailing list archives must be of a considerable size, spanning over a considerable time space.
6. The software development effort undertaken by the community has to be significant.
7. The community must be more than *chasing tail-lights*.
8. The technology developed by the community has to have a broad appeal.
9. Important decisions and arguments within the community have to be conducted on the mailing list.

The final choices

After going through the potential cases and eliminating those not satisfying the criteria I had drawn up, I was left with three candidates. At this stage I realized it was more a question of making a choice that I was comfortable with, than coming up with the ideal case study. The only way to see if a case was suited was to make a choice and start working on it. Being a Linux user, my only two real alternatives were the Linux kernel or the KDE desktop environment. At the time KDE was regarded as the killer application that would wrestle the desktop hegemony from Microsoft. KDE was a tempting choice because of this, even though it did have certain aspects of *chasing the taillights* of Microsoft's Windows desktop. The Linux kernel however did look like a much more interesting case study because of my interest in operating systems. In addition to being a case study of hacking in action, I assumed there were technical sides to the discussion that would prove interesting. The Apache Web server did not tempt me much at the time. It was undoubtedly the most widely used Web server on the Internet, but it had its vocal insider representatives. I very much thought that choosing Apache would render my thesis irrelevant, as some of these more vocal representatives would already have published any results I would find.

My first choice of case study was therefore the Linux kernel. Linux is a hacker project that has gotten enormous amount of attention. It is a technically complex project, with contributors from across the globe. There is ample access to a mailing list for Linux developers, so the source material seemed to be in order. Yet, after a period of study it was becoming apparent that the mailing list did not track any major decision making processes. After a month or so I got in touch with one of the leading Linux developers, David Miller, asking him how the decision making process was handled within the Linux community. He replied that most of the important decisions were actually being made on private e-mail between a handful of central developers (personal e-mail October 5 1999).

It had become apparent that the Linux kernel was not suitable as a case study. It did not fulfill one of the more important criteria: no important decisions were being carried out on the mailing list. While not entirely back at square one, this set me back quite some time and I had to select a new case. At this time the KDE desktop project was no longer as hot as it had initially been. It was riddled in a licensing issue discussion over the QT widget library. QT is the core technological basis of the KDE graphical user interface. It is a commercial cross-platform C++ library for creating GUIs. It had a license that allowed developers of KDE to use QT without licensing fees. This, combined with the project's *chasing taillights* factor, put me off. Despite my initial reluctance, I ended up choosing Apache for my case study.

Selecting the material

Based on my initial work on the Linux kernel mailing list, I realized getting through the sheer amount of e-mail stored in the new-httpd archives would be too time consuming. I had complete archives of the new-httpd mailing list from March 1995 up to today. The amount of data made me look for a way of making the archives more accessible. The Apache project stores their mailing list archives as large text files, each file containing all the e-mails sent through the mailing list over the duration of a month. The format made reading the archives extremely hard. To amend this situation I chose a tool called Hypermail. It processes mailing list archives into a set of Web pages. The amount of e-mail to read was still immense, and I chose to do some initial statistic analysis of the mailing list traffic in order to locate time slices to concentrate my efforts on and to identify possible prime movers within the development group.

Limiting the data

The new-httpd archives are complete from the inception of the mailing list in March 1995. To limit the data material, I drew a line a December 1999. I would concentrate on the data material within these two dates. The choice was influenced by the Apache developers' migration to the Apache2 architecture, an entirely new code base that I had no knowledge of. It seemed like a convenient place to stop as I had then tracked the life cycle of the first generation of the Apache Web server.

Hypermail

Hypermail is a mailbox to HTML converter with threads support. It takes a file in Unix mailbox format and generates a set of cross-referenced HTML documents. Hypermail is itself free software licensed under the GNU General Public License, available from http://www.hypermail.org. The application creates a single HTML file for every e-mail in the mailbox being processed. Each file created contains links to other articles, so that the entire archive can be browsed in a number of ways by following links. Each file generated contains (where applicable):

- the subject of the article,
- the name and email address of the sender,
- the date the article was sent,
- links to the next and previous messages in the archive,
- a link to the message the article is in reply to, and
- a link to the message next in the current thread.

In addition, Hypermail will convert references in each message to email addresses and URLs to hyperlinks so they can be selected. Email addresses can be converted to mailto: URLs or links to a CGI mail program.

For every mailbox being processed, Hypermail generates four index pages. Three are index listings sorting the e-mails by date, subject and author respectively. The fourth index page is chronologic, tracing the flow of discussions in a threaded model. This fourth index provides a good overview of the discussions taking place. When filtered through Hypermail each month is represented with a complete index over the messages sent to the list, with links to the individual messages. In addition to providing the number of e-mails sent onto the mailing list, the HTML pages generated by Hypermail allows the user to browse the archive by thread, author, and date. It is therefore easy to read the mailing list traffic, as in the number of e-mails processed on a monthly basis; the number of active participants on the list, as in who sends e-mail onto the mailing list; and the major participants on the list, as in how many e-mails is sent by each participant.

Statistic analysis of the archives

To get an overview of the mailing list activity, I drew a graph showing each month on the X-axis and the number of e-mails posted to the mailing list on the Y-axis.

The intention of this graph was to get an indication of the effort spent in developing the Web server over time. Iterating between reading the mailing list archives and looking at the graph, I hoped to get some insight into the ebb and flow of the development effort. The underlying assumption is that the introduction of new technology and/or the emergence of a disputed topic would increase the number of e-mails sent onto the list. As such I could use the graph as a guide for locating interesting points in the history of the Apache project.

I also generated yearly graphs for better granularity. The graphs can be found in the appendix.

To gain an impression of the distribution of activity within the Apache group, I drew a graph showing the individual developer's activity measured in the number of e-mails sent to the mailing list pr. month. Using the total number of e-mails sent to the mailing list that month, a figure provided by Hypermail's index listings, I calculated how many percent of the total traffic each participant contributed with. Together with a closer reading of the archives, I hoped this would provide me single out the prime movers within the community. The assumption behind the diagram is that active participants in the development effort will contribute more on the mailing list. As it turned out, this assumption is somewhat wrong. The diagram favors vocal participants, and a closer reading of the mailing list would prove there is not necessarily a correlation between being vocal and the amount contributed. The diagram did show useful as an initial indication of the community's prime movers.

To make these pie diagrams more easily readable I chose to reduce the number of entries by collecting all developers having posted less than 10 messages to new-httpd within the space of a month, into a separate category *Other*. I was also hoping that this would be to provide me with an indication of whether active participants in the development effort stuck through or that there were a rapid change of developers. It could provide me with an indication of whether there is a core of developers sticking through the entire process, and other key developers joining the team for shorter or longer periods of time. Such an analysis could give me an indication of how knowledge is being kept in the development team, whether it is being kept there through tradition—that it is passed down by key members—or some other mechanism.

A correcting to the analysis above is the number of discussion threads each developer has started pr. month. The assumption is that the most active participants are those who initiates discussions. The assumption is that the number of threads a single participant has started, can provide an indication of that participant's effort in pulling the process forward. This analysis will always have to depend on the results of the mailing list's traffic analysis, as an implicit assumption here is that a low volume of traffic indicates the loss of interest by the participants. As with the total messages pr. month figure above, the total number of e-mails sent pr. individual developer does not give an accurate description of each individual developers' activity in the project. However, combined with the number of threads started, a fairly accurate picture of who contributions may be found.

These pie diagrams can be found in the appendix.

Choosing material of interest

The statistic analysis never did yield the expected results in crystallizing certain periods as particularly interesting. It did show a steady increase in traffic on the new-httpd mailing list, though. The only real hint the analysis gave me, was a slump in activity during May to July 1995. The significance of this slump did not occur to me until fairly late in working with this thesis. At first disregarded it. Instead I chose a new approach to finding interesting material. I chose a two angled approach into the mailing list material. The first angle into the material was based on getting an overview of the mailing list archives. I simply started skimming through the monthly indexes generated by Hypermail, starting with March 1995. I looked closer into the particular of topics that seemed particularly interesting, especially long discussions (measured in the number of e-mails in the discussion thread), and recurring subjects. The assumption is that both recurring topics and long discussion is significant to the project participants as they put a lot of effort into the topic. By making notes, and rereading older archive indexes, this approach provided me with a number of episodes and topics that proved interesting.

The second angle into the material was more selective. Based on the project time-line, I looked closer into periods around major releases.. In addition I used a clue provided by the article *Open Source as a Business Strategy* (Behlendorf 1999). The article briefly mentions a controversy between America Online and the Apache group during December 1996. The assumption here is that these are periods requiring decision making. For a release decisions have to be made as to what goes into the release. The AOL controversy would require the group to consider its alternative approaches to the problem.

In order to get an inside view of a hacker community, I arranged two formal interviews with Stig Bakken. Stig Bakken is a prominent member of the PHP Core Team. PHP is a very successful hacker project, and Bakken is a local hacker. Since I already knew him, I set up two interviews with him during the first six months of working on the thesis. The aim of these interviews were to break the hermeneutic circle, and try to find an approach to the material at hand. Since PHP also comes with a plug-in module to Apache, Bakken knew some about the Apache project.

While I do not cite Bakken anywhere in the thesis, his influence on my understanding of the Apache hacker community is significant. I do not agree with some of the views he expressed during the interviews, but they helped making me aware of issues that I would otherwise have ignored.

Based on my two angles into the material and supported by the interviews with Stig Bakken I collected a number of episodes. At first these episodes spanned in time from March 1995 to the end of 1999. As work progressed, I started limiting my scope. Episodes were removed one by one. This process of elimination was iterative, in that episodes were taken out and later reinserted. This process of selecting the particular episodes to choose continued all to the end of working on the thesis. The final choice was a results of the dynamics between data and theory. I leave it with that for now, as the role of theory is discussed later in this chapter.

The final choice of cases are actually limited to a very small time slice, March 1995 until August the same year. Central here is actually the slump in activity that showed up in the initial statistic analysis. It would prove central in that the slump indicates a breakdown in the Apache project. In the final choice of episodes I end up breaking two of the criteria set down when choosing case study. While initially concerned with studying the case over a long period of time, I end up with a time slice of less than half a year. During this period, the development group is still fairly small, counting less than twenty individuals with only a handful of active participants. It turned out that these criteria were not necessary to show the point I was getting to.

Discussion

The interpretative research tradition is based within phenomenology and hermeneutics. The most fundamental principle of hermeneutics is the hermeneutic circle. It is a meta-principle upon which all hermeneutic principles stem from. The circle can be summarized as:

> Thus the movement of understanding is constantly from the whole to the part and back to the whole. Our task is to extend in concentric circles the unity of the understood meaning. The harmony of all the details with the whole is the criterion of the correct understanding. The failure to achieve this harmony means that understanding has failed (Gadamer 1976, p. 117).

The whole context of a phenomena can’t be understood by simply adding up the fragments making up the phenomena. The phenomena must be understood as a result if the fragments and the whole.

My study of the new-httpd mailing list iterates between separate fragments and the context. The fragments are most commonly single e-mails, but can also be discussion threads that together form a larger whole. The whole is in these situations the entire mailing list archives. At other times I regard the single thread a number of e-mails belong to as a whole. The aim of this approach is to achieve an understanding of the Apache project by “iterating between considering the interdependent meaning of parts and the whole they form” (Klein and Meyers 1999, p. 72). Despite this, I am not entirely true to the hermeneutic approach.

In addition to iterating between fragments and the context by reading the new-httpd mailing list archives, I also chose the statistic approach in an initial effort to break the hermeneutic circle. The amount of source material made available by the mailing list archives seemed unsurmountable, and initial attempts at simply reading the archives had given few results. The use of statistics is to be understood as an attempt at organizing a complex and unorganized mass of e-mails. But it is also a failure in a part on the application of a pure hermeneutic approach to the source material.

The basic tenet of phenomenology is that all we can ever know are phenomena, as there is no such thing as a *thing in itself*. Once the phenomena is understood correctly, we know all there is to know (Galliers 1992). In practical terms this means that in order to fully understand the phenomena, the entire context must be grasped. I try to provide a sufficient context for the Apache case study. Chapter 5 is the direct historical background for the Apache project. The historic backdrop for the case is the history of hacking as briefly laid out in chapter 4. As most participants in the Apache project have an academic background within computing. Some of the participants have even worked as professional software engineers. Software engineering, as laid out in chapter 2, therefore plays a role in the context of the project. Despite this, the entire context by which to understand the project is not provided. There are avenues left unexplored. I have not, for instance, tried tracking any of the other related on-line fora like news and IRC where the Apache hackers participate in discussions about Web and Apache.

Interpretative research acknowledges that even the subjects under study are active interpreters. Interpretative theory “suggests that the facts are produced as part and parcel of the social interaction of the researchers with the participants”(Klein and Meyers 1999, p. 74). I have of practical reasons chosen to solely work with the new-httpd mailing list archives. Apart from two interviews with Stig Bakken, I have had no direct interaction with the hackers in general nor any active Apache hackers. By doing so I leave out the subject under study as an active interpreter. This is unfortunate from an interpretative point of view. Only half the story is told. The Apache hackers’ interpretations of the events is never heard. When I have chosen to work in seclusion, it is at the danger of missing out on important interpretations by those actually involved. There is no dynamics between my interpretation and the Apache hackers’, and I have therefore had no correction of my own interpretations. This makes my interpretations more vulnerable than they would otherwise have been if I had included the Apache hackers as active interpreters.

Chapter 7. Vignettes of software systems development

1995: The Internet is still predominantly a command line-based medium. While the World Wide Web has been around for the better part of a year, its adoption is still fairly limited. On-line communities almost exclusively use e-mail, Usenet and Internet Relay Chat to communicate. File sharing is predominantly FTP-based. Predating the Web is Gopher. Gopher is a file sharing service in between FTP and the Web. It employs clickable links to download files. In 20/20 hindsight, almost a decade down the line, it may seem obvious the World Wide Web will succeed. In 1995, though, its success is not that obvious. In fact, due to its widespread use, Gopher seems better positioned to become the next generation Internet information system.

The Apache team is distributed across the United States and Europe. Its members use the Internet to communicate. Their primary communications channel is the new-httpd mailing list. They use the mailing list for announcements and discussions about the Web server's development. As e-mail is an asynchronous mode of communication, there are always a number of discussions running in parallel on the mailing list. Threads of discussion overlap in time and topic. At times several threads merge, but more often a single thread forks into several separate sub-threads. In turn, these sub-threads run in parallel, overlapping each other in time and topic, and at times even merging again.

Browsing the mailing list archives may prove confusing. There are discussions within discussions. Threads are dropped and left unfinished, to be picked up the next day, several months later, or never again. While the archives store e-mails on a monthly basis, and Hypermail giving the possibility of sorting messages by either author, subject or date, the sheer wealth of information present makes it difficult to extract interesting data. To isolate the data foundations for this thesis, I choose to present vignettes from the new-httpd mailing list. Each vignette is a limited selection of e-mails bound together by topic and/or time. To preserve the flow of events and illustrate the complexity of issues the developers have to work with, I first present the vignettes with minimal explanation recognizing that many details may be unfamiliar to the reader. Each vignette is then followed by a commentary that provides some background information about the topics discussed and the vignette itself.

Making decisions

We enter the story upon the last day of February 1995, and the new-httpd mailing list participants are in the process of setting up the project that will be known as Apache a few months later. Until now the mailing list participants have merely been a group of disgruntled users of the NCSA Web server. Early morning local time February 28 Robert Thau posts a declaration of independence to new-httpd (Thau, posted on the new-httpd mailing list February 28 1995). It is becoming apparent that the NCSA team is not going to update their Web server, a situation that calls for action. If the NCSA team isn't going to continue developing their server, someone else has to take over.

Thau has only recently gotten involved with the new-httpd crowd. With a Harvard degree in Biology from the mid-eighties, he is at the time a graduate student by MIT's Department of Brain and Cognitive services. He is working on robotics and various sorts of neural network modeling. Thau will in some ways become the Apache group's mad scientist, constantly experimenting with new features and playing with different approaches to solving old problems, exploring the emerging Web technology. For now, though, he is concerned with announcing the new-httpd mailing list's autonomy.

Even though Thau's declaration of independence is the first e-mail archived in the new-httpd mailing list archives, it is obvious from the subject field and its contents that Thau's message is the conclusion of a longer discussion. The e-mail is a summary of the new-httpd crowd's collective sentiments. It is a mix of a declaration of independence and an action plan. At the crux of the matter is the synchronization problem that stems from having no central authority to organize, synchronize and distribute the plethora of patches available for the NCSA Web server release `1.3`.

> However, 1.3 is not a perfect product, and in the absence of further visible development from NCSA, a lot of people have found themselves fixing or extending 1.3 to meet their needs --- in the process fixing the same bugs and deficiencies over and over, at different sites. (Thau, posted on the new-httpd mailing list February 28 1995)

As the NCSA developers seem uninterested in synchronizing the existing patches for `1.3`, the people on the new-httpd list will take on the task themselves to create a revised version of the software with all popular patches included. Things have gotten out of control, and the new-httpd participants are taking matters in their own hands. In fact, Thau says that the biggest problem facing the Apache group will be the announced beta of the NCSA Web server version `1.4`. Will the new-httpd group have the right for life, a justification for their dissident activities, if the NCSA group finally gears up and releases a new version of their software?

At this stage, though, the project is very much a group of individuals sharing their individual work. People are not sitting around waiting for someone else to take the initiative. The whole effort is, from a software managerial point of view, a complete chaos where anyone is more or less free to do whatever pleases them. Things are so uncoordinated that even active participants on the mailing list are having problems keeping track of the activity. There is no control over what is being worked on, and who is doing what. There are even several servers being used as patch repository independent of each other. Robert Thau's work is based around his own site where he collects patches, while Cliff Skolnick has announced the `hyperreal.com` FTP site as central repository. Despite this, which has its obvious problems, the project seems to be progressing nicely.

The amount of work organizing the patches is quite massive. Often several patches solve the same problem in different ways and depending of different combinations of preceding patches. This makes interdependency between patches at best unclear, at worst mutually exclusive. As new patches rely on the combination of patches applied to the code, the situation is becoming increasingly complicated by the day. After a fortnight Cliff Skolnick posts to new-httpd:

> OK. I think we are getting ourselves into a bind here and loosing track of which patch does what, which is the latest, etc. I think we needs some formal process (ick) for voting patches in or out and stuff. More importantly we need to link the patch reports and bug reports to the discussions somehow, which will not be an easy trick. (Skolnick, posted on the new-httpd mailing list March 15 1995)

Skolnick is a former Sun Microsystems employee and long-time low-level Unix/C programmer. Together with Brian Behlendorf he founded Organic Online, a San Francisco Web design consultancy, the year before. They had found the only real Web server alternative, the NCSA server, lacking in both functionality and stability. Trying to improve on this, they became the main driving force behind setting up the new-httpd mailing list. Robert Thau proposed a loose process in his declaration of independence, but now Skolnick finds the team in the same bind as previous to setting up the mailing list, a situation he wants to change.

In his e-mail, Skolnick proposes a three phased process to deal with the problem. The first phase consists of creating a patch. All new patches are assigned a unique identification number. Once the patch has been submitted, it enters the second phase which consists of peer review and bug fixing. Editing a patch is handled centrally, and every day a change log is distributed to show the past 24 hours' activity. Bug reports are posted to the mailing list. When posting a bug report the affected patch's identification number must be included for reference, and only one patch is allowed pr. bug report. This is done for easier retrieval of patch information from the mailing list archives. Each week a round of votes will be made to determine which of the finished patches shall go into the main code base. Once a patch passes the vote, it enters the third phase: integration. It is now integrated with the main code base. Even though not explicit in Skolnick's e-mail, it is implied that integration is handled by a designated individual who cleans up the code and does a proper integration job. Skolnick's central argument for such a process is that it is "a big part of software engineering as opposed to hacking" (Skolnick, posted on the new-httpd mailing list March 15 1995).

Robert Thau doesn't completely agree. While he agrees with a minimum of traceability in patches and that peer review is important, he is worried that the project will loose momentum. Based on his experience with the NCSA developers, Thau argues the results of too slow a development organization. Having learnt the importance of rapid updates from his own situation with the NCSA group, he does not want the Apache to get into a situation where every minor bug fix has to go through the review process, impeding the update frequency. He is willing to forsake some of the advantages of an administrative system for the sake of speed "... many of the items on the functional enhancements list will take some discussion to decide if we want them, and in what form, and I don't think those discussions should be allowed to hold up the project" (Thau, posted on the new-httpd mailing list March 9 1995). Rob Hartill supports Thau's sentiments: "Our goal as I see it at the moment is to maintain some momentum in the process of discussing ideas and adding them to what was NCSA `1.3`" (Hartill, posted on the new-httpd mailing list March 21 1995).

The actual development organization to emerge from these initial exchanges of opinions is somewhat of a compromise. Rob Hartill, proponent of the need for speed, is actually the first to make use of the voting system (Hartill, posted on the new-httpd mailing list March 15 1995), and even Robert Thau agrees to the need for patch identification (Thau, posted on the new-httpd mailing list March 15 1995). The rest of Skolnick's proposal is never found necessary, and therefore never effectuated. The project does not seem to suffer in momentum from this introduction of a more rigid process, though. Within less than two weeks of the declaration of independence Cliff Skolnick announces his plans for releasing a first alpha version of the Apache Web server (Skolnick, posted on the new-httpd mailing list March 12 1995). A week after that Rob Hartill has the first pre-release of Apache ready for the group (Hartill, posted on the new-httpd mailing list March 19 1995).

Commentary

The *patch* is central to this vignette. A patch or a patch file is the difference listing produced by the `diff` program. `diff` compares the contents of two files. One file is the base which changes to the second file are found. The output of `diff` can be stored in a file. This file is called a patch, so called because its contents can be used to update a copy of the base file with the tool named `patch`. A patch is relative to the contents of the base file. Herein lies the problem facing the Apache developers. The situation is better explained using an example.

Say we have a file, F. Brian finds the bug B_1 in F, and fixes it. This revised file is called F_1. Using diff he creates a patch P_1 that he shares with fellow developer Cliff. Robert finds another bug B_2 in F, and fixes it. Let's denote Robert's revision F_2. Using diff he produces the patch P_2. Cliff has applied patch P_1 to F. Working with F_1 he realizes there is yet another bug in the the code. Cliff fixes the bug resulting in a new revision F_3. Using diff, Cliff produces patch P_3 that he shares with Brian and

Robert.

There are now three revisions of the original file. F_3 relies on the revision F_1, while both F_1 and F_3 stem directly from the original file F.

The problem arises when Robert wants to apply the changes contained within P_3 to his F_2. He first has to apply P_1. With trivial code, this is really not a problem in itself. The problem arises when P_1 and P_2 have changes in a common segment of lines. These conflicts will have to be resolved by merging the contents of the two patches. This usually has to be done by hand. Another problem would be if Robert wants the changes from P_3 but not P_1. Such a situation may prove a bit more tricky, and made even worse if the code is complex or difficult to understand. The task is at best time-consuming, usually fraught with pit-falls, and in worst case impossible to follow through as P_3 relies too deeply on P_1.

The Apache team experienced as situation where numerous patches had inter-relationships that were difficult to chart. For them it was becoming next to impossible to share the many patches to solve the original NCSA code's many bugs.

The non-forking server

It is mid-March and Rob Hartill posts his third e-mail to new-httpd that day. After having been quiet the first half of March, traffic is once again picking up on the mailing list. A `0.1` release has been out for a while, but the team is still discussing the project's name and its ultimate goals. They have therefore spent the past days discussing non-technical issues like the project name and a mission statement.

Rob Hartill is becoming increasingly famous for his work on the Internet Movie Database, IMDb. For him, IMDb is a past-time project, but it was actually this past-time that got him his current job at the Los Alamos National Lab in New Mexico. Hartill is one of the first people to employ the emerging Web technology to make an interesting application. For him the movies are only instrumental, his real interest lies in the challenge of doing something interesting with the new medium presented by the Web. Despite having a day job, Hartill puts some 20 odd hours a week into Apache and IMDb, making him a substantial contributor to the Apache project (Gosh 1998).

By mid-March there are technical discussions on the agenda again. Posting his third e-mail that day, Hartill asks about the problems involved with implementing a non-forking server. He has prototyped two different approaches himself, neither of which seems to be particularly difficult nor complex. His first approach sets up N processes to handle incoming requests. When a process accepts an incoming connection, a new process steps up to wait for the the next connection. His second approach makes use of the BSD socket library's own scheduling mechanism when several processes simultaneously accept connections to the same port. A parent process is used to avoid race conditions where several processes processes accepts the same connection. Hartill suspects there might be issues with both approaches, but he can't see any himself (Hartill, posted on the new-httpd mailing list March 14 1995).

There might be, Cliff Skolnick replies (Skolnick, posted on the new-httpd mailing list March 14 1995), a problem with the way BSD sockets implements multiple accepts to a single port on multi-processor systems. At least with Solaris `2.x` and SGI, there is. There is no defined behavior for multiple accept calls on multi-processor systems. This is not entirely true, corrects Rob McCool who programmed the original NCSA Web server:

> The BSD spec doesn't define it, but all BSD implementations take care of this well. This includes SunOS, HP-UX, AIX, IRIX (even MP), OSF/1, and BSDI. (McCool, posted on the new-httpd mailing list March 14 1995)

Besides, the problem isn't merely isolated to multi-processor systems. McCool is experiencing the same issues on both single- and multi-processor systems running on the Solaris operating system.

Another general problem with non-forking, replies Robert Thau (Thau, posted on the new-httpd mailing list March 14 1995), is the danger of memory leakage. This isn't a problem with forking servers as the handling process always dies after returning its reply to the client. Any memory leakages in the handling process are cleaned up as the process dies. With non-forking the processes do not die, and memory leaks accumulate. These are issues pertaining forking servers in general.

Thau also mentions a problem specific to the Apache code (Thau, posted on the new-httpd mailing list March 14 1995). As the original code is forking, it has no concept of a per-transaction status. Because the handling process dies after finishing off the request, there is no need to reset variables holding values specific for the request. A non-forking server will have to clear all transaction-specific variables before a new request may be handled. There is a third approach to non-forking, concludes Thau, the approach taken by Rob McCool in the NetSite server. Unfortunately, there are issues with this approach as well.

Rob McCool implemented the original NCSA Web server code back in 1994. He has since left the NCSA team, and is now employed by Netscape. At Netscape he develops the company's Web server, NetSite. McCool has kept a rather low profile on the new-httpd list, his follow-up on Thau's e-mail being his first posting that month. McCool confirms both the memory leak and Thau's objections with the NetSite's servers non-forking code. "If I remember the NCSA code right, it leaks memory and file descriptors like a sieve. strdup() is used in a couple of places, though most of the code opts for the stack-based approach of memory allocation. At the time, doing all of that made sense. . . " (McCool, posted on the new-httpd mailing list March 14 1995), continuing with a description of the pros and cons with the mechanism employed by the NetSite server.

NetSite uses what McCool calls a mob process approach to non-forking. Robert Thau describes the mob process approach as “a bunch of processes all doing accepts on the same socket” (Thau, posted on the new-httpd mailing list March 14 1995). Long requests tend to starve the resources, McCool says confirming Thau’s objections to the mob process approach. As processes are not freed fast enough to handle incoming requests, the server may become unresponsive under heavy load. The mob process approach is to start all available Web server processes upon startup. This is done to avoid the shared memory requirements associated with coordinating growth and reduction of a process set. Requests that can’t be handled when there are no available processes, are queued by the operating system. However, there is a limit to the number of connections queued before the operating system starts rejecting incoming connections. This, McCool concludes (McCool, posted on the new-httpd mailing list March 14 1995), is better than an uncontrolled spawning of new processes.

A few days later Brandon Long replies. Brandon Long has taken over McCool’s old position as Web server developer with the NCSA team. Long, like McCool, has kept a low profile prior to this discussion. Now, however, he joins in to explain the non-forking approach taken by the next release of the NCSA server (Long, posted on the new-httpd mailing list March 16 1995).

Like the mob process mechanism, Brandon Long has opted for forking a number of child processes at startup. Unlike the NetSite design, he has chosen to let the parent process act as scheduler. It hands off incoming connections to one of the child processes, before returning to accept new incoming connections itself. This approach is prone to the same limitation as NetSite’s approach, in that long requests starve the resources by not freeing processes fast enough. To overcome this, Long lets the Web server revert to a fork-and-die state under heavy load. The Web server continues to operate in this state until one or more of the initial processes are available again. It is a slight improvement of the original NCSA code.

Over the next few days a short discussion ensues over the NCSA `1.4` non-forking. Rob Hartill suggests serialization before a call to accept (Hartill, posted on the new-httpd mailing list March 16 1995), but Rob McCool argues that this will indeed make the parent process a bottleneck as handling one request will require three process switches (McCool, posted on the new-httpd mailing list March 17 1995). Hartill replies that his suggested system is only required for multi-processor systems, as it has the potential of staying one step ahead of each connection (Hartill, posted on the new-httpd mailing list March 17 1995). With that the thread of discussion dies without having reached a conclusion. No patch to implement non-forking is submitted, and the issue stays dormant for another two weeks.

Late March and early April the NCSA team releases two beta versions of their upcoming `1.4` series of Web servers. Rob Hartill is not particularly impressed with the series' non-forking mechanism. He says it is "fundamentally flawed" (Hartill, posted on the new-httpd mailing list April 9 1995). Through stress testing, Hartill has discovered a concurrency issue with `1.4`'s non-forking code. Despite Hartill's objections, Robert Thau goes through with one of his many side projects, and merges the NCSA `1.4` non-forking mechanism with the Apache code.

Thau restarts the non-forking discussion again in mid-April by suggesting to release a beta of Apache with the NCSA `1.4` code. His only objection to a new Apache release with the non-forking code is that it will be using NCSA `1.4` code before the NCSA team has released their `1.4` server publicly (Thau, posted on the new-httpd mailing list April 12 1995). Apart from certain reservations regarding legalese, the NCSA team does not mind this (Long, posted on the new-httpd mailing list April 12 1995) (Frank, posted on the new-httpd mailing list April 12 1995). Brian Behlendorf, on the other hand, is not too keen on releasing a non-forking Apache yet (Behlendorf, posted on the new-httpd mailing list April 12 1995).

Returning to the issues Hartill has expressed about the NCSA's non-forking code, Behlendorf would like to wait with releasing a non-forking Apache. Instead of serializing the processes, which is the approach taken by Brandon Long in the NCSA code, Behlendorf would rather do multiple accepts on a single port. He believes this to be a better approach if the multi-processor issues can be resolved. Brandon Long is apparently not too happy with his non-forking code. He joins the discussion to back Behlendorf's argument (Long, posted on the new-httpd mailing list April 12 1995).

Once again the discussion sputters and dies without there being reached a conclusion. When Apache release `0.6` comes up for votes, there is no mention of non-forking code.

Commentary

The vignette shows the number of concerns involved with implementing non-forking behavior in the Apache server. It also shows the complexity with which the issues intertwine, the sheer amount of caveats to take into account, and the depth and breadth of technical knowledge required to understand the problem complex. In addition to directly related technical concerns, the developers discuss indirectly related topics like interprocess communication, licensing and the morality of using other people's code, to mention a few subtopics. The main topic of discussion is the best non-forking architecture for multiple software and hardware platforms. The challenge is to come up with the most effective architecture for non-forking that works equally well on single- and multi-processor systems.

Berkley sockets, BSD sockets, or simply sockets is a TCP/IP implementation originally written for BSD Unix in 1983. Sockets, provides both kernel level support by implementing the TCP/IP stack and an abstract programmers' interface (abbrev. API) to program networked software. Its story is not unlike that of Apache. Without going into details, the code was made publicly available on the Internet leading to widespread use by most Unix versions. Many Unix versions started with some version of the Berkley networking code, including the API. These are often called *Berkley-derived implementations*. As such the API's function calls are a *de facto* standard when developing networked application for Unix (Stevens 1998). The problem is, as shown in the vignette, slight differences in behavior between the various derived sockets implementations.

The issue of forking vs. non-forking is not an issue particular to Apache, nor to Web servers for that matter. It is an issue related with Berkley sockets and the sequential nature of single process applications. "To allow for many processes within a single host to use TCP communication facilities simultaneously, the TCP provides a set of addresses or ports within each host" (Postel 1981). The server listens to a specific port. The server process is ready to accept incoming connections with a call to the socket API's accept(2) function. The accept is blocking, first returning the incoming socket's file descriptor when a peer connects to the port. As single-threaded applications are sequential, the server can't return to accepting new connections until it is done processing the connection just having been made. As long as the accepting process is handling a connection, no new connections can be accepted and the server may seem to be non-responsive. For low-traffic, quick request–response cycles, such a mechanism is acceptable, but it does not scale at all under heavy load.

In multi-tasking operating systems like Unix, concurrency is used to resolve the problem. By spawning a child process as a new connection is accepted, control of the newly accepted connection can be handed to the child process. The parent process immediately returns to accepting new connections as soon as it has handed control to its child. The child processes the connection, and dies when the connection terminates. Spawning new processes is handled by the Unix system call fork(2). Handling multiple requests this way is consequently called forking.

The main drawback with forking is the overhead involved. The system call creates an exact copy of the original process. The contents of the parent's memory area is copied into a new memory area. While some Unix version have optimized the forking process, the overhead involved is still substantial. Forking is therefore a suboptimal solution when it comes to performance. Forking a new process for each incoming connection may become a bottle-neck under heavy system load and/or in situations where great many connections are made in a short time-span. This is the background for discussing non-forking with Apache.

To avoid the overhead associated with on-demand forking, the non-forking server will fork a number of child processes upon startup. These processes are pooled. There are several strategies to manage the pool's processes. The best strategy for pooling is the real topic of the non-forking discussions on new-httpd. The limiting factor with choosing the right design for non-forking is that Apache is supposed to work on a number of Unix versions, and on single- as well as multi-processor systems. It is the combination of multi-processor systems and sockets derivatives that are giving the Apache team a challenge. As Rob McCool notes, the behavior of serialized accepts on multi-processor systems has no specified behavior for various Unix versions' sockets derivatives.

Concurrency is another problem with developing for multi-processor systems. Single-processor systems maintain an illusion of concurrency by splitting processor time between the processes. Processing is still sequential, although performed in small time slices. Synchronization is therefore possible by defining volatile sections. A volatile section is a code segment that can't be shifted out of the processor until it is done processing. It is a good way of avoiding race conditions in concurrent code on single-processor systems. On systems with multiple processors, several processes actually do perform in parallel. Volatile sections can't ensure against race conditions, as code that leading to the race condition may actually be running in parallel on another processor. This increases the complexity of concurrency and synchronization on multi-processor systems.

It is this synchronization that seems to be leading to the problems Rob Hartill notes with Brandon Long's original non-forking code in NCSA `1.4`.

Virtual hosting

Randy Terbush has the patch for a new feature:

> Something I have not seen discussed on this list is the possibility of adding a "Multi-homed" support feature. I have a *small* patch as implemented by Bob Baggerman that I could place on hyperreal. Would the group prefer that I contact Bob and ask him to submit the patch himself?. (Terbush, posted on the new-httpd mailing list March 10 1995)

It is a fairly busy day on the new-httpd mailing list, and Terbush' e-mail seems to be lost in the general activity. Administrativia is pretty much the order of the day. A total of 11 messages are submitted in a discussion about the project's name. Another 2 messages deal with setting up a home page for the project. Cliff Skolnick has set up a script that sends an e-mail to the list every time a patch is submitted or updated, generating another 19 computer generated messages in addition to the 28 sent by the project members. For whatever reason there is no reply to Terbush' message, and no patch is submitted.

A few weeks pass by, but Randy Terbush seems not to have given up on the idea of *multi-homed* support:

> I was just beginning to hack on multi-homed support. The changes are fairly minor and I wanted to ask the group what their thoughts are about implementation.
>
> The direction I would like to take would be to reset ’DocumentRoot’ based on the answering IP address (ie the IP that the server is answering for). I see it being much like ’ScriptAlias’ such that if the IP contacted is 200.100.50.1, DocumentRoot gets set to /www/docs/200.100.50.1.
>
> Comments?
>
> The other implementation I have seen sets a CGI environment variable to SERVER_ROOT which requires an index.cgi to be in the toplevel document directory. Seems like all of those execs could add up to a lot of overhead. (Terbush, posted on the new-httpd mailing list April 1 1995)

Cliff Skolnick has gotten a head start on Terbush. He has implemented this feature already, but not yet submitted the patch due to an unfixed error with logging (Skolnick, posted on the new-httpd mailing list April 1 1995). Skolnick’s patch is submitted a short week later on April 6 (Terbush, posted on the new-httpd mailing list April 6 1995), and slightly updated by Randy Terbush the next day (Terbush, posted on the new-httpd mailing list April 7 1995). The patch is not met with immediate enthusiasm. David Robinson doesn’t like the way it is implemented, nor how it is being configured (Robinson, posted on the new-httpd mailing list April 9 1995). He argues that one should instead run several httpd processes, each listening on a different IP address. This would require no changing of the code, instead utilizing the features provided by the sockets API. The nasty side effect of this, as Cliff Skolnick points out, is that this requires having several independent configuration files (Skolnick, posted on the new-httpd mailing list April 9 1995).

Skolnick’s second objection to running independent httpd processes is based on a future projection. He says “when non-forking is standard” (Skolnick, posted on the new-httpd mailing list April 9 1995), think of the consequences running separate processes will have on performance. Running all within a single process and its child processes makes the Web server scale better as resources may be shared between the virtual hosts. Robert S. Thau steps in to further support Skolnick’s argument for running all in one process. Their argument is based on experience: “Given the peakiness of Web loads, most processes in a server pool are likely to be idle most of the time --- they’re there anyway so the server doesn’t have to fork, and get even slower, in response to transient load peaks” (Thau, posted on the new-httpd mailing list April 9 1995).

As for Robinson’s objection to the user-friendliness of Skolnick’s virtual host patch—“Using virtual hosts requires an understanding of the DNS and TCP/IP that most people don’t have. The current patch really confuses the issue by pretending that it is selecting on the hostname the user used, whereas it is really selecting on the ip-address the browser uses” (Robinson, posted on the new-httpd mailing list April 9 1995)—there are no replies to it. The format will remain the same throughout the entire lifetime of Apache 0.x and 1.x series.

When votes are up for release `0.5` on April 8 (Hartill, posted on the new-httpd mailing list 8 1995), the virtual hosting patch's reception is still luke warm. While there are no vetoes for the patch, it still hasn't gotten sufficient votes to make it into the next release. In the nick of time Brian Behlendorf submits his votes (Behlendorf, posted on the new-httpd mailing list April 8 1995), and virtual hosting makes it into the `0.5` release. While some consider virtual hosting a strategic important feature in competition with other Web servers (Terbush, posted on the new-httpd mailing list April 8 1995), the majority of the Apache developers seem indifferent.

Commentary

Under the surface of this vignette, there is an argument between aesthetics and practicalities. Robinson's objection over the feature is based on a minimalist approach to programming. Apache has a command-line option to specify the location of the configuration file. The configuration file has an option to determine what port to bind the Web server to. The operating system itself supports running several processes of an application. Robinson's argument is one of the approach of least effort, a practical application of the *keep it simple stupid* philosophy (Raymond 1999b). While having practical implications, the *keep it simple stupid* philosophy is really one of aesthetics. It is reflected in the credo "good programmers know what to write, great programmers know what to rewrite (and reuse)" (Raymond 1999, p. 33). Reuse requires having understood not only the problem to be solved, but also existing technologies.

Skolnick's argument is of a more practical mind, but also contains elements of reuse. His primary argument is one of resource management. A single process approach to virtual hosting allows several Web sites to better make use of the same resources. Having several processes doing the same thing running on a same system is a waste of resources. Instead of competing over the resources, the Web sites can make use of the possibility that different Web sites have different traffic peaks. Skolnick's secondary argument is one of convenience as well as reuse. If the same configuration file can handle several virtual hosts, settings common to the hosts may be shared and thereby reused. Settings specific for a virtual host is found in the configuration part specific to that virtual host. Having the entire configuration in one file is considered advantageous as it saves time having to find and open several files when configuring a virtual host.

Crisis, chrysalis, Shambhala

Late April, the Apache project is scant two months old, but cracks are starting to appear. "You can add honey to shit, but you only get sweet tasting shit", Cliff

Skolnick vents his frustration after a long night of hacking (Skolnick, posted on the new-httpd mailing list April 26 1995). It is a few minutes past 5 in the morning when Skolnick sends his e-mail. “I’m getting more and more frustrated trying to wedge stuff into a large program that was not well architected . . . I am not proud of the patches I need to do, but there is no clean easy was of integrating the junk”. He is not the only one frustrated with the turn of events: “Some time soon we’ll need to take a look at redesigning Apache completely”, Andrew Wilson adds (Wilson, posted on the new-httpd mailing list April 26 1995). It is apparent that new features are becoming increasingly harder, to the point of impossible, to integrate with the existing code base. “It’s [the Web server] been growing organically and while individual new components are well designed they *are* bolted on to a rather mixed up program that could benefit from a comb and cut” (Wilson, posted on the new-httpd mailing list April 26 1995).

Since release `0.5` on April 8, getting sufficient votes is becoming increasingly harder for each release. Comes May, and the voting system starts breaking down as there is simply not sufficient activity for anything substantial to be voted on. Hartill is able to gather sufficient votes for release `0.6.3`, but upon starting work on release `0.6.4` in mid-May it is becoming increasingly apparent that he is fighting an uphill battle. Hartill is determined to follow through with new releases of the Web server. By now there are simply no new patches to vote on. Brian Behlendorf later describes Hartill’s fight as “shoving an elephant down the eye of a needle” (Behlendorf, posted on the new-httpd mailing list June 30 1995). The lack of patches doesn’t reflect the Web server’s state. The server has serious technical problems.

Early May sees the release of Apache `0.6.3`, but the release is later skipped because an incompatibility problem with W3C’s Arena browser. The browser is not able to properly content negotiate with Apache `0.6.3` (Thau, posted on the new-httpd mailing list May 5 1995). Then a problem is discovered in with handling of CGI scripts in an earlier release, `0.6.2`. The issue sorely needs fixing, but nobody rises to the occasion (Hartill, posted on the new-httpd mailing list May 11 1995). `0.6.2` is in turn discarded in mid-May due to this unresolved issues. A new bug-fix release, `0.6.4`, is released in mid-May. It is meant to resolve the situation. Yet, the problems are not over. In quick succession bug fix versions `0.6.4b` through `d` are released over the next few days. Amidst all this, there are still unresolved issues like the XBITHACK, unresolved since early April(Behlendorf, posted on the new-httpd mailing list April 7 1995), and no decision has been made as to Apache’s non-forking behavior.

A statistic analysis of new-httpd speaks clearly of the problems. The volume of traffic on the mailing list is halved between April and May. By June the traffic is only one fourth that of April. Activity is at an all time low, and traffic on the new-httpd mailing list will never again drop this far. By mid-May development activity seems to have ceased almost completely. Hartill is practically keeping the project alive single-handedly. He alone is responsible for 26% of the mailing list's total traffic, having started 44% of all threads.

Throughout May and June Rob Hartill is trying to sort out the problems with the release `0.6.x` series on his own, while also working on a new release, Apache release `0.7`. The release cycle slows down as people stop voting. It is obvious that the Apache development effort is quickly going stale. Within less than 6 months after breaking free from the NCSA group, Apache seems fast heading into the same bog of problems with infrequent updates as the NCSA team suffered under. When Apache `0.7` is released (Hartill, posted on the new-httpd mailing list May 29 1995), it is the result of Rob Hartill's hard work. The development effort is no longer a group effort. Looking back Randy Terbush recalls that "Rob Hartill has ... done a great job of biting the bullet and pulling `0.7` into a publicly consumable form." (Terbush; posted on the new-httpd mailing list June 30 1995). Yet, the `0.7` release cycle is not without its own share of problems (Behlendorf, posted on the new-httpd mailing list June 6 1995; Cox, posted on the new-httpd mailing list June 7 1995; Watkins, posted on the new-httpd mailing list June 8 1995).

Mid-June, and Robert Thau shares his garage project with the rest of the team:

> Over the past few weeks, I've been off on a garage project to explore some possible new directions I thought *might* be useful for the group to pursue. ... What I have right now is a server I'm calling Shambhala. ... The basic idea of Shambhala is to make a modular "tinkertoy" server, to which people can easily add code which is valuable to them (even if it isn't universally useful) without hairing up a monolithic server. (Hartill, posted on the new-httpd mailing list June 13 1995)

Not only is Thau trying to modularize the Web server, his rewrite even takes measure to remove some of the old error sources. Thau's code is still missing a great deal of Apache's extra functionality, and it is fairly unstable. A week later, and the Shambhala code is both more stable and has some more functionality added.

Working in parallel with Thau's Shambhala project, is an effort to make the existing code more stable. `0.7.2` is released first week of June, only to suffer the same fate as `0.6.4`. A series of rapid bug fix releases `a` through `k` are unleashed over the next two weeks. Lead by Hartill, work on the original code is facing serious problems: "I just grabbed a copy of 0.7.2d, compiled it under Solaris 2.4 w/ gcc and it still seems to have a problem I've seen before - random links, usually images, hang, and the error log shows an socket error and then dead children", Ryan Watkins reports (Watkins, posted on the new-httpd mailing list June 8 1995). The server's memory footprint is too large (Hartill, posted on the new-httpd mailing list June 16 1995). Trying to fix it a new problem arises: "well the suckers are down to 260k each at startup (was 548k), so that's promising.. trouble is that a SIGHUP (restart) causes the new children to die when servicing any request" (Hartill, posted on the new-httpd mailing list June 17 1995).

After two weeks of frequent bug fixes to the `0.7.2` release, `0.7.2k` is abandoned as it still does not handle redirection correctly. The next day Rob Hartill releases `0.7.3`, which fixes the redirection problem. Later the same day release `0.7.3` is already at bug fix release `c`. There seems to be no end to the problems. “Apparently there is an fd leak happening somewhere... and killing the children doesn’t seem to free them, only killing the parent”, Brian Behlendorf reports (Behlendorf, posted on the new-httpd mailing list June 20 1995). “I’ve had a quick look at apache 0.7, and it sees to suffer from a major design flaw, namely that it always starts up a fixed number of processes.” David Robinson reports (Robinson, posted on the new-httpd mailing list June 26 1995).

Interest in Apache is gone: “Feature development on Apache 0.7 has been static ... I don’t remember anyone reporting that they were working on any ideas/code.”, Rob Hartill sighs (Hartill, posted on the new-httpd mailing list June 30 1995). Despite its unfinished state, an increasing number of people are moving from Apache to Shambhala. The excitement is returning:

> Has anyone else in the group taken a look at RST’s Shambhala? The code is ultra clean. Truely non-forking. Predictable. Fast! (Terbush, posted on the new-httpd mailing list June 29 1995)

Suddenly the mailing list bursts into activity. Rob Hartill is wondering if this means the demise of the Apache project (Hartill, posted on the new-httpd mailing list June 30 1995). At the very least he wants to know how Shambhala relates to the Apache project (Hartill, posted on the new-httpd mailing list June 30 1995). Over the next few weeks the original code base that Hartill has put an enormous amount of effort into, is scrapped and the Shambhala code replaces it as the official Apache code from release `0.8`. For a while Thau replaces Hartill as the project’s central authority. Unlike Hartill, Thau doesn’t have to pull the project by himself. Even though Thau decides what goes into the Shambhala code and what doesn’t, a great number of developers contribute with bug fixes and several developers are porting their features to this new modular architecture.

Commentary

Hartill took upon himself the unenviable job of pulling the Apache project forward when nobody else seemed interested in participating any longer. Robert Thau assumes, with the other developers’ consent, the role of a benign dictator. While Thau is given credit for the Shambhala code, Hartill is actually keeping the project together during the times of hardship. Both make most important decisions on their own. The difference is that Hartill does so out of necessity, while Thau does it based on an argument of architectural consistency.

On the technical side, the Shambhala code implements only a bare bones Web server just serving Web pages. The request handling is split into several stages. Each stage has a hook where extra functionality can be added to the server. This extra functionality is implemented in modules. This modular design allows the addition of new functionality without having to change the core Web server code. That new features had to be wedged into the core Web server code had been the original problem with the original NCSA based code. With Shambhala this major source of errors had been eliminated, in the process allowing experimental code to be added to the Web server as modules without everybody having to apply this experimental code. If technology is society made durable, Shambhala is the Apache community's wish for both an experimental and production quality Web server in one, made durable.

Towards Apache `1.0`

By August activity on the new-httpd mailing list is higher than ever before. From release `0.8.5` on August 1, Apache reaches release `0.8.12` by the end of the month. Activity is heating up. On August 8 the Apache group releases the first public beta of Apache. The response is overwhelming. After the beta release a whole host of new developers join new-httpd. With 24 developers active on the mailing list in July, the number increases to 33 by August. After the voting system broke down during the May-June crisis, it still has not been revived. Robert Thau retains his position as project coordinator with the power to decide which patches to include and which to exclude. As the new design is coming together and the API stabilizes, Thau relinquishes control. On August 22 he sets up the first ballot in over a month. While the response is not overwhelming, it is sufficient for the vote to go through. Even though new blood has found its way onto the new-httpd mailing list, only the original group members—people like Brian Behlendorf, Rob Hartill, Roy Fielding, and Randy Terbush—that cast their votes.

The upsurge is only temporal. By September activity has dropped again. It is still high, though, but nowhere near the August peak. Once again the Apache group seems stuck in an endless myriad of bug fixes. Last month's rapid release cycle has lost speed. Despite only allowing bug fixes and no enhancements into the code, there are only two releases during September (release `0.8.13` on September 11, and `0.8.14` on September 20). It is obvious that the Apache project is heading straight into the same problems it had during April and early May. By October list activity is dwindling again. Only during the May-June crisis has activity been this low. Once again Rob Hartill sees the need for him to take charge. He had inquired about the switch to a `1.0` release number in late August (Hartill, posted on the new-httpd mailing list August 29 1995), with no response from the other developers. On October 2, as a follow-up to his first inquiry, Hartill proposes a release schedule for an Apache `1.0` release (Hartill, posted on the new-httpd mailing list October 2 1995). Instead of driving the process forth through coding, as he did with the `0.6` and `0.7` releases, he suggest a timetable of for getting the `1.0` release ready. While meeting little resistance, the idea does not meet an overwhelming response either. Some agree; most keep their silence: "...continued silence is a sign of apathy or satisfaction." Hartill writes sarcastically (Hartill, posted on the new-httpd mailing list October 5 1995).

As soon as plans for the `1.0` release are announced, the developers start planning ahead on the next release after `1.0`. Ben Laurie is especially active with this. He proposes an operating system independence for Apache. Such an undertaking would require another rewrite of the code to abstract the OS dependent elements of the code. But it is not simply the issue of Apache running on other platforms that interest the Apache developers. Instead of adding new features to the existing code base, Roy Fielding would like the group to concentrate on its usability in form of commenting the code and developing proper documentation (Fielding, posted on the new-httpd mailing list October 15 1995). In the mean time Ben Laurie goes about adding support for Secure Socket Layer (abbrev. SSL) to Apache (Laurie, posted on the new-httpd mailing list October 16 1995). As with Garey Smiley's OS/2 port, SSL/Apache forks off the original code base. I.e. it is not integrated into the Apache code base. Instead SSL/Apache maintains its own code base independent of future changes to the Apache code.

Parallel to this activity, Rob Hartill is pushing forward on the `1.0` release. Once more he is fighting an uphill battle, and his original timetable slips bit by bit as the other developers show little or no interest in the `1.0` release. On October 17 release `0.8.15` the one that according to Hartill's timetable will be released as `1.0` is made public (Thau, posted on the new-httpd mailing list October 17 1995). Upon having the release candidate, Hartill sets out preparing the `1.0` release further. He rounds up a handful of developers who can provide binaries for the different Unix platforms (Hartill, posted on the new-httpd mailing list October 18 1995). A proper release should not have to be compiled by the end-user. Then the whole release plan crashes. Internal dissent in the Apache group crushes Hartill's efforts. As it turns out, a faction within the group feels the code has not been properly tested (Thau, posted on the new-httpd mailing list October 24 1995). On October 26 Hartill cannot be bothered any more. He has been fighting the uphill battle too long, with no response from the group. "I'm not going to suggest any more timetables for 1.0 'cos they just get ignored or rejected" (Hartill, posted on the new-httpd mailing list October 26 1995). Instead of a `1.0` release, a new bug-fix release is made public November 5 (Thau, posted on the new-httpd mailing list November 5 1995), setting a definite end to Hartill's `1.0` release effort.

Perseverance and persistence is the key to success. Hartill has not altogether lost hopes of a `1.0` release. In mid-November he makes a new effort (Hartill, posted on the new-httpd mailing list November 14 1995). The `1.0` release has been over 3 months in the waiting by now. This time around the other developers respond positively to Hartill's effort. Several people involve themselves with clarifying the outstanding issues for the release. Show stopper bugs are traced, patches to be included are announced, and a date for the code freeze is set. Suddenly the project explodes with activity. After a longer discussion, it is determined which modules to include with the binary distribution of Apache (Terbush, posted on the new-httpd mailing list: November 25 1995). An SSL version of the `1.0` release is being prepared , and on December 2 the Apache group can announce the release of Apache `1.0`. Within less than a year the Apache project has matured from being simply a collection of much-needed patches for the NCSA Web server, to having its own architecture and API, having reached a `1.0` release, and being the second most used Web server on the Internet.

While development activity on Apache has been somewhat up and down, the figures of use tell a different story. Apache is becoming increasingly more popular. In August of 1995 the NCSA and Berners-Lee's CERN Web servers are topping the usage statistics with over 75% of all Web servers on the Internet running either one of those. Apache is third on the statistics, with over 15% less users than the CERN server. Only 3.47% of all Web servers on the Internet is running Apache at this stage. From there on, the usage statistics only go one way: upwards. During September, October, and November Apache remains third most used Web server. Its market share increases to 11.39% in November. During the same period both the NCSA and CERN servers loose ground with less than 65% market share by November. Then, in August, Apache has passed the CERN server, and is now the second most used Web server on the Internet with a total market share of 17.91%. During this period, the NCSA server, still the most used Web server on the Internet, has dropped from a hegemony of 57.16% market share in August to 37.71% in December. Things are looking very bright for the Apache developers. (Source: The Netcraft Web Server Survey http://www.netcraft.co.uk/Survey/Reports/)

Chapter 8. Analysis

A number of topics are raised in the vignettes. In this chapter I will try to analyze them and relate them to the theory.

Software systems development

The development process employed by the Apache project is a compromise. Early Apache developers seems to participate by one out of two reasons. There are those who see the project as a way of getting the number of patches under control. For them the project's function is more of the administrative nature. Without a centralized body, patch dependencies get out of control and it will become impossible to fix the server's bugs. On the other hand, there are those who regard the project as a means of exploring and expanding the Web technology. They want flexibility and ease of adding new features to the server.

The patch and vote system that emerges from the initial discussions, aim to provide an administrative framework for tracking and integrating patches into the server while retaining speed and flexibility. The process reflects the tension between control and flexibility inherent in the project.

The patch and vote system consists of two elements: revision control and quality assurance. The necessity of revision control is undeniable. Pre-Apache NCSA patches bear witness of that. The lack of a fixed baseline to base patches on, results in a dependency chaos that eventually becomes too difficult to resolve. By rolling a number of patches into a release, the team provides a common baseline to work towards. The latest release always being the baseline to base patches on. Frequent releases, one of Robert Thau's main concerns, makes patch dependencies manageable. The fewer patches not yet integrated with the code base, the fewer inter-dependencies to resolve when applying a patch.

Patches are voted for integration with a release. Before a patch may be integrated, it will have been tested by several other team members. The regular flow of events is for a developer to fix a bug or add new functionality to his own Web server. If it seems to be working, he runs diff to create a patch. The patch is then shared with the rest of the development team. Instead of developing extensive test suites, the developers run newly patched up code on high-traffic Web sites. Experience from running the code results in a vote for or against integrating the patch.

It may seem, then, that Eric Raymond is correct. Is the Apache project just about sharing code and parallelizing the debugging effort? To understand the project's strength, I believe we have to look beyond the paraphernalia of the implementation effort.

Use driven development

Models and methods is not the kind of terminology that suits the work style of the Apache project. Describing the project’s working style with such a terminology would be to force the language of software engineering onto the software development practices of the Apache group. The patch and vote system may be construed a process, but is rather a technique to manage the complexity of virtual distributed work. That the whole patch and vote process is abolished while Robert Thau completes the Shambhala core supports this. None of the Apache developers are ever explicit about the project’s work style beyond the patch and vote system. The work style is implicit and often fairly difficult to see even for the developers involved with the project. This is at odds with literature on software systems development. Erling Andersen, for instance, says that “We have emphasized that prototyping must not be confused with an unsystematic and unplanned work style. If prototyping is to succeed, it has to be driven along explicit guidelines and by people who are experts in the area [of prototyping].” (Andersen 1998, p. 350, my translation). Booch, Jacobson, and Rumbaugh 1999 explicitly states that:

> There is a belief held by some that professional enterprises should be organized around the skills of highly trained individuals. They know the work to be done and just do it! They hardly need guidance in policy and procedure from the organization for which they work.
>
> This belief is mistaken in most cases, and badly mistaken in the case of software systems development. . . . developers need organizational guidance, which . . . we refer to as the “software development process” (author’s note: process as in the convergence of several methodologies). (Booch et al. 1999, p. xvii)

Apache does not have any *organizational guidance* which they work by. Their work style is *unsystematic and unplanned*. Yet, they obviously succeed in their efforts—having almost 60% share of the Web server market a good indication of success. I argue that this sheds some light on the over-emphasis on the perniphernalia of methods, models, tools and techniques in the software industry. Software engineering, where the quartet above play the lead roles, belongs to the *scientific management* tradition where the work process is rationalized. “The basic premise of scientific management is that one can reduce the best way to do a given job to a set of instructions and give those instructions to someone who does not know how to do it independently but who will then be able to do the job by following the instructions,” (Orr 1996, p. 107). This *set of instructions* is the model and methods of software engineering. Within a sociological tradition this reduction of the job to a set of instructions is viewed as the management’s effort to get control of the knowledge involved in the work process, de-skilling the labor. By de-skilling management can hire less skilled and cheaper labour and as such rationalize the organization (Braverman 1974).

The central element in developing Apache is *use*. The roles of user and developer converge. Use plays an apparent role in testing and quality assurance. Through a brute force approach where the software is put in use, bugs are detected and new functionality tested. Raymond calls this parallelized debugging. Bugs are broadcasted to the other developers through the mailing list. But more important is the role use plays in knowledge creation, something the May-June crisis shows.

Managing complexity

The Apache project grows out of a situation where the complexity of change has grown out of hand. Patch interdependencies had become unmanageable to a certain extent. Thau and Skolnick’s process discussion is to a large extent about handling the complexity of change. They each represent two disparate ideas on how the complexity is to be managed.

Skolnick proposes administrative mechanisms to handle the complexity. The core of Skolnick’s system is a unique identification number assigned to every submitted patch. The id is to be used when discussing the patch, and every day an automated e-mail is to notify the developers of changes made to the patches. Once again the id is to be used for traceability. Skolnick’s plan is to manage complexity by linking the patch, discussions about the patch, and possible bug reports the patch is to solve. His strategy, which is common in formal development environments, tries to reduce the environmental complexity of change by introducing traceability. Traceability suggests a controlled environment, where every change comes as a result of a planned course of actions. This can be as the response to a bug report, or a feature request. Both are based on the notion of the change request which controls what to be changed and often how. Through strict control of the changes to be made, Skolnick’s administrative process is to reduce the complexity of change.

Thau’s proposal suggests quite the opposite. By letting go of any administrative control on changes, he suggests to control the complexity through scope reduction. Instead of adding constraints to the changes, Thau wants to frequently base line the system. In this context each new release, informal or formal, is a baseline which all unresolved patches must use as basis. The administrative complexity of Thau’s suggestion is marginal. Instead it distributes the complexity of change to each person submitting a patch. In theory, a new release may invalidate all previously submitted patches. As a patch is relative to its base file, once the base file has changed the patch no longer applies. Not every file in the software will be updated between releases. And for those updated, only patches directly dealing with code blocks that have changed will require substantial work for bringing it up to date. For other patches, it is just a question of inserting the changed code into the new release of the file, and run diff anew.

The end result, though, is a mix between Skolnick’s high ceremony approach and

Skolnick's bare bones approach. A voting model is introduced for quality assurance, and all patches are assigned an identification number for reference. The rest of Skolnick's model is never effectuated.

Where Skolnick's proposal is centralized, Thau's is decentralized. This is the major difference in ideology their proposals. Seen from the point of view of rationalizing and making the development effort more efficient, Thau's suggestion sees a lot of double work as patches need to be updated after a new release. By controlling the changes to be made, Skolnick's approach reduces and perhaps even eliminates this redundancy. Thau's proposal suggests that the work lost in updating patches is far less than the amount of time spent administrating a change control system. He sees control as a limiting factor on the creativity and speed of the development effort, and is willing to spend time updating patches to retain these advantages.

Thau's proposal sheds some additional light on Eric Raymond's principle "Release early. Release Often" (Raymond 1999). While Raymond lists this principle as an enabling condition for innovation, the Apache project also shows it as an effective means of handling the complexity of change.

There is not much discussion surrounding neither Skolnick nor Thau's process proposals. Instead the project participants start using those parts of the two systems they fit. This way of voting with their feet can be interpreted as a variation on Levy's hands-on imperative. Through their actions the project members decide which elements of a process model is required. While this might be a valid interpretation, it lends much of Levy's romantic view on hacking, ignoring the nuts and bolts of everyday software systems development. A system is required to manage the complexity of change, which previous to setting up the new-httpd mailing list has been completely out of control.

Another aspect of the complexity issue, an aspect largely ignored by process and methodology literature, is the complexity posed by technology. Technical complexity is not trivial. The non-forking server vignette shows that even though Apache is developed for Unix operating systems, small implementational differences between the various operating systems and hardware differences impose a substantial technical complexity. The Apache project employ two strategies to resolving the complexity. One approach is analytical, the other uses brute force.

Upon implementing both non-forking behavior and virtual hosting in the Apache server, a group of team members discuss the various aspects of low-level network behavior for BSD sockets. Approaches are suggested, and responses given based on experience and knowledge of operating system specific behavior. This approach to managing technical complexity depends on in-depth knowledge of the combinations of technologies, knowledge that stems from experience using this technology. It is a social activity that is based on building a shared understanding of the complexities faced by the technology.

Once the discussion has reached a certain point, it is replaced by the brute force approach to managing technological complexity. The brute force approach is to implement the functionality and then try it out in as many different operating environments as possible. The many combinations of operating system and BSD implementation does prove difficult, as many of the bug reports during May-June-July deal with networking errors on certain platforms. The brute force approach is then to fix the bug, and try once again. It is a process of trial and error. Trial and error is frowned upon by efficiency afficiados, as it is regarded as a failure to fully analyse the problem at hand. The question to be raised is whether a discussion could have fully exhausted every possibility and every difficulty in the combination operating system and networking implementation. Another question is whether taking the discussion all the way and fully explore every possibility would be more time-efficient. It is difficult to say, and it is a hotly debated issue.

The above discussion may give the impression that the Apache developers manage technological complexity in a well-defined manner. It is not always so. The introduction of Shambhala, for instance, sees no prior discussion. Here Robert Thau just goes straight ahead and implements the new architecture. A brute force approach, of sorts. Discussions on the appropriateness of the solution comes more as an afterthought. Even with virtual hosting the real work starts with implementation. Unlike Shambhala, implementing virtual hosting is more of an iterative process where analysis and brute force follows in succession a number of times. Still, the two elements of managing technological complexity remains: analysis and brute force.

May-June crisis

During May and June 1995 the Apache project is on the brink of dissolution. Rob Hartill is the main, at times even the single, driving force behind the `0.6` and `0.7` releases. For each release in these series, there are immediate bug fixes. It might seem that whenever a bug is fixed, new bugs appear in another end of the system. The problems Skolnick complains about during work on the `0.5` release, remains. Why is that?

Software development aspect

I see several ways of reading the crisis story, depending on which view of software systems development is chosen.

Booch, Rumbaugh and Jacobson says a software development process “defines *who* is doing *what when* and *how* to reach a certain goal” (Booch et al. 1999, p. xvii). Apache’s patch and vote process states what is to be done, how and when, but it doesn't say anything about who. An element is missing from Booch et al.'s equation of process. There are no responsibility matrices. Work and responsibilities are those of the team as a collective. Can this short-coming in the process be the reason behind the May-June crisis? The question to be answered is whether a responsibility matrix would have solved the problem at hand? The problem seems to be an organizational one, rather than a short-coming with the project's process model.

Bezroukov speaks of the *problem of the lowest hanging fruit*. The problem is described accordingly:

> Those who can program naturally tend to work on programs they find personally interesting or programs that looks cool (editors, themes in Gnome), as opposed to applications considered dull. Without other incentives other than the joy of hacking and "vanity fair" a lot of worthwhile projects die because the initial author lost interest and nobody pick up the tag. (Bezroukov 1999)

This comment is consistent with the findings of Eric Monteiro in his study of the Internet Engineering Task Force's IPng work-group (Monteiro 1998). Monteiro's findings pointed towards the fact that once the challenging issues had been dealt with, activity stopped even though there was work left on the project before it could advance to the next step of the IETF work process. The outstanding issues were of a mundane character, most of them simply hard work that had to be done. Yet, no one took hold of these issues to pull the project forward, leaving the IPng in a half-finished state for a long time.

The Apache project is a volunteer effort. There is really no way of leveraging any pressure to make people submit new patches and fix bugs. If volunteers don't want to contribute any more, they leave. If enough volunteers stop contributing, the collaboration breaks down, something which happens with the Apache project during the May-June crisis. A volunteer project like Apache balances on a very fragile exchange economy. Eric Raymond (1999) argues that participants in volunteering efforts like the Apache project must feel they are getting a sufficient return on the time and energy invested in the project. In a technical project like Apache, the return value is easiest measured in form of the technical quality of the software, but there are also social elements involved, Eric Raymond points out. When the returns on the investment drops below a certain level, people stop contributing. With this is mind, the May-June crisis may be understood as a symptom that fixing bugs is becoming so difficult or tedious that contributors to the Apache project doesn't feel they are getting return on their investments.

That Hartill has been shoving "an elephant down the eye of a needle" and that Robert Thau does the Shambhala rewrite for himself are both examples of the single individual's role and importance in software development projects. These findings are consistent with the relative participation by team members commented upon in (Curtis et al. 1993, p. 72). Describing the group dynamics of their case study Curtis et al comments how the a small number of individuals dominate the design process. Like Hartill and Thau in the Apache project developer D1 of Curtis et al.'s study impacted upon the team effort both technically and administratively as he, like Hartill and Thau, tried to seek out and integrate new technical knowledge into the project. D1, like the Hartill/Thau duo, also influenced the team process by taking the initiative in the administration of team duties. Where D1 routinely volunteered to coordinate group activities, Hartill volunteers to pull releases `0.5` through the entire problematic `0.6` series single-handedly. Thau, on his side, volunteers for the other difficult issue, that of rewriting the code to allow easier integration of new features. The findings concerned with the diagnosis of the May-June crisis can therefore be said to be highly consistent with similar studies of software development efforts.

Returning to Raymond's exchange economy argument, it may be used to explain the May-June crisis in another way. Maybe it isn't the promise of bug free software that attracts contributors to Apache in the first place, but the chance of enhancing the Web technology. The state of the code makes adding new features too difficult, and people feel they aren't getting back sufficient on their investments. Reading the mailing list reveals that the server functionality remains fairly stable from the introduction of Shambhala to Apache `1.0`. This sustains the idea that bug fixing is the only thing left on the agenda. Is NCSA based Apache code sufficiently stable? The sheer number of complaints on the `0.6` and `0.7` release series' bugs and the number of bug fix releases to these series speaks of a server code that is far from sufficiently stable to be deployed in a production environment. Rob Hartill even discourages people from using these series in production . Now that there isn't much new to add to the server, the incentive to fix bugs is lost.

This line of argument is akin to Eric Raymond's scratching an itch argument. Raymond says the main motivation for contributing to a volunteering effort is to scratch an itch, i.e. solve a problem or need the contributor has himself. His argument has been criticized for ignoring the community factor of an open source project. The idealized image of hacking is that of contributing to the common good. For the Apache project, the May-June crisis shows that Raymond is right. Apart from Rob Hartill, nobody seems particularly interested in participating in doing the dreary work of bug fixing.

Whether it is becoming too hard to fix bugs or that there is no incentive left to fix them is hard to say. It isn't unlikely that it might be a combination of the two factors. With this in mind, it is hard to think how a better defined *who* factor in the Apache process model would have prevented the May-June crisis. It seems more an organizational short-coming, as there is no way of making people into doing anything. Would another process have solved the problem? There really is no data to sustain any view on the matter. It would obviously have been easier to forced people into fixing the bugs, but there is no telling if the bug problem would have remained. Would any other process have been able to foster the amount of innovation seen thus far in the Apache project? Again, without a comparative study there is no telling.

Of course, the whole line of reasoning above hinges on the assumption that fixing bugs is a boring activity shunned by the Apache developers. Maybe there are other reasons why bug fixes aren't submitted to the project? Looking at the May-June crisis, it seems almost a tailored addendum to Naur's article on programming as theory building. The project's increasing problems with integrating new features and stabilizing the original NCSA based code may seem like a failure on the Apache developers' behalf to grasp the theory behind the NCSA code. It seems to be Naur's compiler story all over again. Judging from reading the mailing list, the source of the project's increasing problems is more prosaic. I think it is safe to conclude that the original NCSA code was poorly written, with little thought as to maintainability in mind. It is, after all, Rob McCool's first attempt at programming C.

What's interesting with the crisis is that from it emerges the next generation Apache architecture, Shambhala.

Direction, directing, drift

Central to the Apache group's existence is the Web server's development. Without a Web server to develop, they are nothing at this stage. That is why they worry what will happen when the NCSA team releases version 1.4 of their Web server. Seen from a more corporate point of view, the the Apache Web server's competitive edge is of strategic importance to the Apache group's survival. Their *raison d'etre* is to create an improved version of the NCSA server based on available patches. If version 1.4 of the NCSA Web server provides the same functionality, or even improves on the functionality already available in the Apache Web server, there is little justification for the Apache group to continue its work. Formulating a strategy is a means of staking out the future direction, and it would be appropriate for a newly started project to stake out its direction so the participants have a common goal to work towards.

The Apache project really doesn't stake out a clear direction.

The traditional approach to strategic information systems, that is information systems that by its competitive edge is of strategic importance to its producer, is to appraise the environment through a conscious and analytic process. Through a conscious and analytic process "threats and opportunities, … the strengths and weaknesses of the organization, key success factors and distinctive competencies are identified and translated into a range of … alternatives" (Ciborra 1994, p. 7). Once the optimal strategy is found, it is laid out and implemented. This way the strategy is made explicit.

It has been pointed out that this way of thinking about strategy is flawed (Ciborra 1994). The central axiom for traditional strategy forming is that anything can be resolved if only sufficient analytical power is applied to the problem. Like non-canonic work practices separates doing from learning, the traditional approach to formulating strategy focuses on abstract knowledge and cranial processes instead of situated knowledge from within the organization. “Strategy formation tends to be seen by the mechanistic school as an intentional process of design, rather than one of continuous acquisition of knowledge in various forms, i.e. learning” (Ciborra 1994, p. 9). The central critique of traditional strategy formation is that it is difficult to plan before the fact, and that competitive advantage stems from the exploitation of unique characteristics within the organization.

The realization that an organization’s strategic advantage lies in exploiting its unique characteristics goes a far way in explaining Nonaka and Takeuchi’s enabling conditions fluctuation, creative chaos and autonomy. Implemented in the organization, the traditional strategy can be understood as a battle plan. In order for it to succeed, the entire organization has to walk in step and work towards a shared common goal.

The Apache group has no explicit strategy, yet they succeed. The closest they are to an explicit strategy is that they want to develop the next generation Web server, reflected in the mailing list’s name: new-httpd. There is no authority to tell anybody what to do, and there is no *battle plan* ordering strategically important features and bug fixes to be implemented. Everyone is free to work on whatever they would like. Instead of viewing this as a potential hazard, it proves to be central in revitalizing the Apache Web server after the May-June crisis. While Rob Hartill is working hard to fix bugs in the original NCSA based architecture, an effort that is as close to seeing clear authority in the development group at this stage, the organization’s freedom allows Robert Thau to implement the Shambhala architecture. With Shambhala the Apache Web server transcends its role as an improved NCSA server, becoming a fully fledged application on its own.

Not only does Shambhala revitalize the stale development effort, it proves to be strategically important as it gives the server the flexibility required for the next generation of Web server. Yet, this happens without ever formulating an explicit strategy. The strategy emerges from the organization’s grassroots, addressing issues important to the developers. There is a crisis where routines and organizational frameworks break down. While the old project, improving the original NCSA Web server, goes on, Robert Thau’s re-implementation introduces redundancy. From the creative chaos a possible solution, Shambhala, emerges. This is possible as all developers enjoy practically unlimited autonomy within the organization.

There is another, less fortunate side-effect of this lack of direction. It is best seen in Rob Hartill’s tireless effort from May through November to get the Web server into a state where it can be labeled version `1.0`. Central to Hartill’s efforts is the development of a server with the stability to serve Web pages without crashing. While head of the Debian GNU/Linux distribution project, Bruce Perens is attributed to have said his job was like herding cats. This is a metaphor that can just as easily be said about Hartill’s effort. While he is trying to push the project forward towards a stable release, other participants seem more interested in side-projects like SSL/Apache. Yet others simply don't see the rush and are quite happy to let the bugs be fixed as people see fit. The problem with getting the code into shape for a `1.0` release is closely connected with Bezroukov's "problem of the lowest hanging fruit" (Bezroukov 1999), but it also shows the kind of drift to be expected from this kind of distributed collaborative work.

The knowledge-creating organization

Instead of looking at the project in terms of having a process model, the Apache project organization can be seen to exhibit Nonaka and Takeuchi's enabling conditions for knowledge creation. The developers' work style isn't defined by what they do, but rather by how they choose to cooperate. Individual work is the basis for the project's collective work. There is no central control over who does what. Individual autonomy is practically unlimited. As the virtual hosting vignette shows, "[o]riginal ideas emanate from autonomous individuals, diffuse within the team, and the become organizational ideas" (Nonaka and Takeuchi 1995, p. 227). Instead of immediate adoption of Cliff Skolnick's virtual hosting implementation, the patch undergoes scrutiny and is subjected to objections. Improvements are suggested, as well as alternative technical approaches to virtual hosting. Even though the initial implementation of virtual hosting is individual work, building a justification of the chosen approach is collective. Instead of looking at development as a relay race consisting of a series of individual works, individual work is followed by cooperation and discussion.

This way developing virtual hosting isn't as much about implementing the feature, but rather about building a theory of the feature. Based on Skolnick's implementation the group builds a theory of the feature. The theory building is only part of the entire knowledge-creating cycle resulting in virtual hosting. Use had shown that it would be fortunate for the same Web server to manage the contents of different independent Web sites. At first the feature is called *multi-homed*, it is later renamed *virtual hosting*. Virtual hosting is the metaphor used by Skolnick and Thau when arguing against Robinson's suggestion to make use of the underlying socket technology to provide the same feature. Instead of multiple Web servers running on the same computer, virtual hosting gives the impression of several servers running. Nonaka and Takeuchi says the metaphor is effective in creating and elaborating a concept (Nonaka and Takeuchi 1995, p. 221), virtual hosting shows how the metaphor helps framing the feature and direct its implementation.

Through collaboration and discussion tacit knowledge is transformed to explicit knowledge. It has an element of "understanding through narration" (Orr 1996, pp. 178-179). It is Robinson's objections, his "intrusion" (Nonaka and Takeuchi 1995, p. 230) into Skolnick's work, that leads to the discussion about why the concrete implementation is to be chosen. Instead of accepting the existing implementation at face value, redundant information about how the same effect could be implemented using existing socket API functionality. As Skolnick's feature works, suggesting a different approach could be considered both destructive and counter-productive. Instead the differing, even competing opinions are used to justify the existing implementation. The degree of redundancy within the project is even more apparent during the May-June crisis. For a while two competing server architectures are being worked on. Rob Hartill is working on stabilizing the existing NCSA based architecture, while Robert Thau is implementing the new Shambhala architecture. A lot of effort is put into both implementations. Nonaka and Takeuchi considers such redundancy "especially important in the concept-development stage" (Nonaka and Takeuchi 1995, p. 230), as it helps transform tacit into explicit knowledge. While it is never articulated, exploring the Web technology is a driving force for many participants in the Apache project. This is tacit knowledge that Robert Thau's transforms into explicit knowledge with his Shambhala architecture. By modularizing the server architecture, new functionality may be added without having to make changes to the Web server's core functionality.

Both vignettes show a distinct flow of events, resembling Nonaka and Takeuchi's five phase model of organizational knowledge. An individual developer's experience triggers the idea for new functionality. The idea is shared with the community, not always leading to more than encouraging replies. The idea is implemented and shared with the community where it is probed, questioned and alternatives are suggested. The probing and questioning is both a way of cross-leveling knowledge (Nonaka and Takeuchi 1995, p. 236), but also an approach to building a shared understanding—a theory—of the functionality. This cross-leveling of knowledge is especially apparent during the non-forking discussion. While discussing Apache's non-forking functionality, developers from both the NCSA team and Rob McCool at Netscape share their experiences with non-forking behavior. Both Rob Hartill and Robert Thau have both implemented non-forking, but the discussion with the other teams spawns a new period of trial and error before they settle upon a solution.

Talking about software

So far the Apache developers' role has not been discussed. The Apache project bears resemblance with participatory design. Through their study of real-life projects, Waltz et. al. observes that for developers to understand users' needs software it is important they speak the same language "or, at least, dialects whose semantics are similar enough to facilitate communication and understanding" (Curtis et al. 1993, p. 63). Unlike the unified process where requirements are captured through use-cases, no explicit requirements gathering takes place in the Apache project. Quite particular to the Apache project is the convergence between user and developer. Developers often play the role of users, like Robert Thau does when using the wish for having simply one configuration file as an argument for Skolnick's implementation of virtual hosting. In other discussions, like non-forking, Thau plays the role of systems expert again.

Unlike the unified process where formal iterations are employed to ensure periodic feedback and communication from users, the Apache project has a continuous communication between users and developers. The feedback is often immediate, but it is not always possible to tell whether the team members are playing the role of user or developer. Often they play both roles, and it isn't all that interesting to talk about the split between developer and user. Julian Orr draws up a triangular relationship between user, technician and machine (Orr 1996). The same triangular relationship can be seen between user, developer and software. It is by talking about the software that developer and user can reach a common understanding of what they are dealing with and the requirements for the software to be developed. It is also through talking about the software with the user once the software is running, that the developer will make an understanding of how the software fulfills its requirements and which are the shortcomings that have to be corrected.

If we look at how virtual hosting came about it is apparent that it is due to the convergence of roles. The need for virtual hosting arose as a user-land problem. However, since both Thau and Skolnick are developers they choose to implement the feature themselves. After having the functionality partially in place, can they start talking about the software and possible directions. But it isn't always necessary to even produce software. Knowledge stemming from use—Robert Thau speaks about the peakiness of Web server load—enriches the technical discussion surrounding virtual hosting. The convergence between user and developer lets user-land considerations effortlessly blend with technical discussion. The triadic relationship remains an important part of the Apache project's work-style.

At the same time as the users of the above example do share their explicit knowledge about their own requirements to virtual hosting and that a possible solution to the problem would stem from combining the users' operational experience with the developers' knowledge of the software and the underlying operating system, it can be argued that the ensuing discussion after Hartill has forwarded the message is in fact a new phase of socialization. "… experienced-based operational knowledge often triggers a new cycle of knowledge creation. … the users tacit operational knowledge about a product is often specialized, thereby initiating improvement of an existing product …" (Nonaka and Takeuchi 1995, p. 72). As it happens, the users' request is technically infeasible and as such does not initiate an improvement of the Web server. Backtracking a bit I will argue that the initial drive for virtual hosting does stem from the combination of experienced-based operational knowledge that has triggered a cycle of knowledge creation. By combining explicit knowledge from the ISP domain and the Web technology, Terbush and Skolnick have found the need of hosting virtual Web sites on one computer. Terbush's sharing of two models for handling this feature and Skolnick's actual realization of the feature are both socialization triggered by combination. This is somewhat at odds with the original model of knowledge creation where the process starts with socialization. I would argue, and I believe the virtual hosting vignette bears sufficient arguments to support my case, that innovation actually starts with combining knowledge from different domains.

In all of this, it may be easy to forget there is a technical aspect to the Apache project. The non-forking discussion is complex and filled with technical nuances. The participants post fragments, assuming knowledge of the issues by the other participants. Through the implementation of possible solutions and discussions, the team builds a joint understanding of the problems, caveats and pitfalls involved with implementing a non-forking server.

Chapter 9. Conclusion

The Apache project shows software systems development as an activity iterating between the collective and the individual. Individial initiative is paramount to maintaining the project’s momentum. Yet, building an understanding of the software, the theory behind it so to say, is a social activity. As important as building the system, is the building of a joint understanding of the software system and its environment through collective reflection and the sharing of experiences and individual insights. With a collective mental model of the system, architectural drawings play a no important role.

While the artefact of architectural drawing is missing, it does not mean there has been no form of initial surveying to base prior to building the software. Apache starts out as a working piece of software, the original NCSA code. The original code proved inadequate in the long run, but it plays an important role in the initial surveying. The NCSA server plays the role of a working prototype. Fixing its bugs and adding new functionality gave the Apache developers something to talk about. While never expressed, the period leading up to Rober Thau’s release of Shambhala can be understood as a period of collective exploration of what the Apache developers really want for their Web server. At the heart here is the dynamics between use and learning, and the importance of the user-developer convergence. It is the individual work of Robert Thau that saves the day, as he catches the essence of what the project members want of their Web server and implements it. It is Thau’s personal insight and his understanding of the Web server that is the Shambhala code. Of course, Thau’s insights relies on the collective understanding built through working on Apache during the first six months of 1995.

The key to understanding learning and innovation in the Apache project therefore lies in understanding the dynamics of the user-developer convergence. Innovation is not a result of detached reflection and planning. It does not come from a thorough, structured analysis of the problem domain. Instead it comes as a byproduct of direct interaction between understanding, creating and using the software. Innovation in this respect becomes understanding what is needed to make use of the emerging Web technology for different purposes. Internet Service Providers have the use for hosting a number of individual Web sites on the same server. From this need rises virtual hosting. PHP rises from the need to provide server side processing to Web pages. The user-developer convergence is learning by doing, and thereby innovating by doing.

The learning dynamics of the user-developer convergence can be explained using Nonaka and Takeuchi’s theory of knowledge creation, as I have in this thesis. Pelle Ehn’s makes use of Wittgenstein’s language game to make a similair point in his study of the UTOPIA project (Ehn 1992), which would be an equally fruitful angle into my material. Both theories say something about how knowledge is shared.

They highlight the social aspect of buidling a shared common understanding of the problem domain. More interestingly for software systems development in practice, is the enabling conditions for such an exchange of ideas to take place. Nonaka and Takeuchi have identified these in their work, but for the special case of software systems development I believe the addition of *use* is appropriate. To understand software prototypes needs to be used, becoming the focal point of an ongoing dialog. With this thesis I have tried to show how succesful such an approach can be.

While much of my analysis focuses on the knowledge side of software systems development, a goal with the vignettes has been to show the techical complexity involved with developing the Web server. While a trivial piece of software in and of itself, the complexity presented by interoperability across hardware and software platforms require a good deal of technical knowledge. While Apache is written to run on Unix platforms based on similar APIs, there are nuances between the platforms that must be taken into consideration when writing code. Hardware presents its own challenges, as an issue like symetric multi-processing for instance, affect the way software libraries behave.

The patch and vote process is an administrative way of managing the complexity of change. Shambhala is a technical sollution to solve some of the same problems, but also as a way of handling the project’s technical complexity. The architecture isolates the server’s core functionality, providing hooks to attach additional functionality. This shows that there are limits to how complex a software system can be, despite all efforts to collaborate on building an understanding of it. It also shows that complexity needs to be handled, and it can’t always be handled in the human world. Software complexity is a technical issue that requires technical know-how to solve. Solving it, on the other hand, isn’t neccesarily the result of reflection, but rather the contrary: the result of practical use.

Hacking, as shown in Apache project, is therefore to be understood as a social endeavour iterating between the collective and individual, emphasising on the skills and insights of individuals combined with collective reflection and sharing of ideas and opinons in an environment with little or no difference between learning and doing.

Future Research

My case study is limited in scope. Compared with seven years of development, I have narrowed the case down to a duration of some six odd months. I have further limited its scope my focusing on the development and knowledge aspects of the development effort. By broadening the scope, there are other pieces of the Apache story that warrants a closer examination. In addition, the Apache story touches upon and hints at aspects of software systems development that might be well worth a closer look. In this section I will try to highlight a few of these tidbits of future research that appeals to me.

The Apache project shows that a traditional software engineering methodology isn't required to develop software. There is no proof to invalidate that methodology may be of use in when developing software, though. It would be ignorant, not to say arrogant, to assume that methodology plays no role in developing software. There is a reason why software engineering has gained such wide-spread adoption. However, proponents of software engineering have their own ways of rationalizing its uses. Methodology can at times seem fuelled by its own inner logic. To look at a project adherring to a formal software development process using much of the same theory and methodology as applied to the Apache project, could lead to some interesting insights into both the nature of software development methodologies and how software is really developed.

Due to its wide spread adoption, Apache is an important part of the Web infrastructure. There are two ways of looking at the infrastructural role Apache plays: a micro scale and a macro scale. At the micro scale it plays an important role in the infrastructure of a Web site. The server is the part that makes the Web pages availble on the Internet. Apart from that, there are features specific to Apache. Many Web sites make use of these features. Making great changes to Apache could mean considerable restructuring of Web sites powered by Apache. These are the kind of scaling issues Robert Thau has to take into consideration when writing the Shambhala core. There are definite elements of scaling of infrastructure in while building the justification for Shambhala, elements which I have chosen not to include in this thesis but which are an important factor in understanding how software systems are being developed.

In the initial draft this thesis included a vignette titled the "AOL debacle" that takes place around Christmas 1996. At that time AOL was the largest Internet provider in the United States, providing Internet access through its own wide area computer network system. Access to the Internet was indirectly provided through Web proxies. An upgrade to their proxy software made all Web sites powered by Apache unavilable to AOL's users. AOL claimed this was an implementational error with Apache, while the Apache group claimed the contrary. The vignette shows both the technical and organizational elements of scaling infrastructure with a large inertia of installed base. A closer examination of the debacle could provide insights into managment of open, cooperative infrastructures like the Internet, lacking a central authority to enforce standards on actors within the infrastructure. There is also a technical aspect to the debacle. It shows some of the transition strategies inherent in the HTTP, an interesting issue for better understanding how to design scalable IT infrastructure.

There is a conflict not only between labor and capital, but also between managment planning and the role of a development department plays. The conflict is apparent in large organizations where development departments are seen as instrumental in implementing the management's strategies. Deadlines are set in context of management plans. Seen in context of a software development department with little or no influence on the decision process, these deadlines seem arbitrary and are often impossible to meet.

The Apache project is a simple organization with few conflicting interests. It has none of the organizational complexities of large, commerical organizations with the constant tension of interaorgainzational interest conflicts. Previous research has looked at role software systems development plays in resolving the conflict between capital and labor, but as far as I can tell there is currently no research on developing software seen from the developers' point of view. It would be interesting to see how software is being developed in a corporate environment of constant intra-organizational tension, what role the development department plays in the power games between differing interests, and which role the developers themselves assume in such an environment.

There is no telling how the Apache project would have turned out if it did employ a more traditional software engineering approach to the development effort. As there is no data to do a comparative studt, it is therefore impossible to do a qualitative analysis of their approach. It is also impossible to tell how the Apache project's approach to software systems development would have worked in another environment, say for developing mission-critical real-time applications. Empiric studies of work practices in more traditional software engineering environments could shed some further light on the role methodologies really play in developing software systems. Of special interest would empirical studies of fairly similair projects with different approaches to software systems development, be. This could shed some light on the role of cannonical and non-cannoncial practices.

Does the Apache provide any evidence that software systems development is not an engineering discipline? It certainly shows that there is both a technical and knowledge element involved. But what about the entire engineering metaphor. Is engineering really applying mathematics and scientific methods to construction? What is the role of construction drawings in an engineering process? Is building houses simpler than software systems development because all parties involved have an understanding, Ryle's theory, of what a house is? Some work has been done on the appropriatness of comparing software systems development with building houses, but can a study of construction builders' work practices shed further light on the appropriatness of the metaphore?

References

Three kind of references have been used in this thesis. The first is the traditional bibliographic references. Secondly, I have used material found on the World Wide Web. These are found together with the main bibliographic references. When referring to Web sources with reliable static archives, like the Internet Engineering Task Force's RFC archives or the back issue archives of on-line publications such as First Monday, I include a link to the Web page in question. Other Web references, like material found on personal Web pages and project pages I include an access date to indicate that the page may have changed since I accessed it. In the main body of the text, when referring direct quotations from Web source, the page number included is only an approximation of the location. The page number included is the number of PageDown I have pressed to arrive at the quotation. This approximation should suffice as on-line quotations are easy to find using the browser's search command.

The third kind of reference used is references to e-mails found in the new-httpd mailing list archives. For traceability I have included these as links to the Hypermail archives I have set up while working on this thesis. Unlike the two other bibliographic reference types, the e-mail references are collected under a separate heading towards the end of this chapter. The messages are sequentially ordered by date instead of by author name. This has been done to illustrate the flow of discussions on the mailing list.

Bibliography

Ivan Aaen, Peter Bøttcher, and Lars Mathiassen, "The Software Factory: Contributions and Illusions", *Proceedings of IRIS 20*, Institutt for Informatikk, UiO, Oslo, 1997.

Jane Abbate, *Inventing the Internet*, The MIT Press, Camebridge, 1999.

Erling Andersen, *Systemutvikling*, NKI Forlaget, Oslo, 1998.

Edited by R. E. Allen, *The Concise Oxford Dictionary of Current English*, Eight Edition, Oxford University Press, Oxford, 1991.

Jørgen Bansler, "Systems Development Research in Scandinavia: Three Theoretical Schools", *Scandinavian Journal of Information Systems*, Volume 1, 1989, pp. 3-20.

Brian Behlendorf, “Open Source as Business Strategy”, *Open Sources: Voices from the Open Source Revolution*, Edited by Chris DiBona, Sam Ockman and Mark Stone, O’Reilly & Associates, Inc, Sebastopol, 1999, pp. 149-170.

Tim Berners-Lee, L Masinter, and M McCahill, *Uniform Resource Locators (http://www.ietf.org/rfc/rfc1738.txt?number=1738)*, 1994.

Tim Berners-Lee and D Conolly, *Hypertext Markup Language - 2.0 (http://www.ietf.org/rfc/rfc1866.txt?number=1866)*, 1995.

Tim Berners-Lee, Roy Fielding, and H Frystyk, *The Hypertext Transfer Protocol -- HTTP/1.0 (http://www.ietf.org/rfc/rfc1945.txt)*, 1996.

Tim Berners-Lee, *Weaving the Web: The Past, Present and Future of the World Wide Web by its Inventor*, Orion Business Books, London, 1999.

Nikolai Bezroukov, “Open Source Software Development as a Special Kind Type of Academic Research (Critique of Vulgar Raymondism)”, *First Monday (http://www.firtsmonday.org)*, Volume 4 Issue 10, 1999.

Gro Bjerknes and Tove Bratteig, “User Participation and Democracy: A Discussion of Scandinavian Research in Systems Development”, *Scandinavian Journal of Information Systems*, Volume 7, 1995, pp. 73-98.

Robert Boguslaw, *The New Utopians: A Study of System Design and Social Change*, Prentice-Hall, New Jersey, 1965.

Grady Booch, Ivar Jacobson, and James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, 1999.

Pierre Bourdieu, *Outline of a Theory of Practice*, Cambridge University Press, Oxford, 1977.

Scott Bradner, “The Internet Engineering Task Force”, *Open Sources: Voices from the Open Source Revolution*, Edited by Chris DiBona, Sam Ockman and Mark Stone, O’Reilly & Associates, Inc, Sebastopol, 1999, pp. 47-52.

Harry Braverman, *Labor and Monopoly Capital: The Degradation of Work in the Twentieth Century*, Monthly Press, New York, 1974.

Fredrick Phillips Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley Longman, Inc, Reading, 1995.

John Seely Brown and Paul Daguid, “Organizational Learning and Communities-of-Practice: Toward a Unified View of Working, Learning and Innovation”, *Organizational Science*, Volume 2 Issue 1, 1991, pp. 40-57.

Christopher B Browne, “Linux and Decentralized Development”, *First Monday (http://www.firtsmonday.org)*, Volume 3 Issue 3, 1998.

Gibson Burrell and Gareth Morgan, *Sociological Paradigms and Organisational Analysis: Elements of the Sociology of Corporate Life*, Heineman, London, 1979.

Lyman Chapin, *The Internet Standards Process (http://www.ietf.org/rfc/rfc1310.txt)*, 1993.

Alan F. Chalmers, *What is this thing called science?*, Second edition, Open University Press, Buckingham, 1978.

Claudio Ciborra, “Change and Formative Contexts in Information Systems Development”, IFIP Conference on Information Systems Development for Human Progress in Organizations, May 29-31, 1987.

Claudio Ciborra, “The Grassroots of IT and Strategy”, *Strategic Information Systems: The European Perspective*, Edited by Claudio Ciborra and T Jelassi, John Wiley & Sons Ltd., Chichester, 1994, pp. 3-24.

Alistair Cockburn, *Agile Software Development*, Addison-Wesley, Indianapolis, 2002.

Michael A. Cusumano, “The Software Factory: A Historical Interpretation”, *IEEE Software*, Volume 6 Issue 2, IEEE Computer Society, 1989, pp. 23-30.

Bill Curtis, Joyce J. Elam, and Diane B. Walz, “Inside a software design team: knowledge acquisition, sharing, and integration”, *Communications of the ACM*, Association for Computing Machinery, New York, Volume 36 Issue 10, 1993, pp. 63-77.

Bo Dahlbom and Lars Mathiasen, *Computers in Context: The Philosophy and Practice of System Design*, NCC Blackwell, Cambridge, 1998.

John James Davies, *On the Scientific Method: How Scientists Work*, Longman, London, 1968.

Edited by Chris DiBona, Sam Ockman and Mark Stone, *Open Sources: Voices from the Open Source Revolution*, O’Reilly & Associates, Inc, Sebastopol, 1999.

Hubert L. Dreyfus, *What Computers Still Can’t Do: A Critique of Artifical Intelligence*, The MIT Press, Cambridge, 1999.

Pelle Ehn, *Work-Oriented Development of Software Artifacts*, Arbetslivscentrum, Stockholm, 1988.

Pelle Ehn, “Scandinavian Design: On Participation and Skill”, *Participatory Design: Principles and Practices*, Edited by Douglas Schuler and Aki Namioka, Lawrence Erlbaum, Ltd., Hillsdale, 1993, pp. 41-78.

Paul Feyerabend, *Against Method*, Third Edition, Verso, New York, 1996.

Christiane Floyd, “Outline of a Paradigm Change in Software Engineering”, *Computers and Democracy: A Scandinavian Challenge*, Edited by Gro Bjerknes, Pelle Ehn and Morten Kyng, Avebury, 1987, pp. 193-210.

Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modelling Language*, Second Edition, Addison Wesley, Upper Saddle River, 1999.

Andrew L Friedman, *Computer Systems Development: History, Organization and Implementation*, John Wiley & Sons Ltd., Chichester, 1989.

H G Gadamer, “The Historicity of Understanding”, *Critical Sociology: Selected Readings*, Edited by P Connerton, Penguin, New York, 1976, pp. 117-133.

Robert Galliers, “Choosing information systems research approaches”, *Information systems research: Issues, methods and practical guidelines*, Edited by Robert Galliers, NCC Blackwell, Cambridge, 1992, pp. 144-162.

Clifford Geertz, *The Interpretation of Cultures*, Basic Books, New York, 1973.

Rishab Ayer Gosh, “Cooking Pot Markets: An Economic Model for the Trade in Free Gods and Services on the Internet”, *First Monday (http://www.firtsmonday.org)*, Volume 3 Issue 3, 1998.

Gisle Hannemyr, “The Art and Craft of Hacking”, *Scandinavian Journal of Information Systems*, Volume 10 Issue 1&2, 1998, pp. 255-262.

Gisle Hannemyr, “Technology and Pleasure: Considering Hacking Constructive”, *First Monday (http://www.firtsmonday.org)*, Volume 4 Issue 2, 1999.

Rob Hartill and Roy Fielding, *Apache voting rules and guidelines (accessed November 13 2002) (http://httpd.apache.org/dev/voting.html)*, 1995.

Rudy Hirschheim and Heinz K. Klein, “Four Paradigms of Information Systems Development”, *Communications of the ACM*, Association for Computing Machinery, New York, Volume 32 Issue 10, 1989, pp. 1199-1216.

Jan Rune Holmevik, *The History of Simula (accessed November 13 2002) (http://java.sun.com/people/jag/SimulaHistory.html)*, 1995.

Christian Huitema and Phill Gross, *The Internet Standards Process -- Revision 2 (http://www.ietf.org/rfc/rfc1602.txt)*, 1994.

Tove Håpnes, “Not in Their Machines: How Hackers Transform Computers into Subcultural Artefacts”, *Making Technology Our Own?: Domesticating Technology into Everyday Life*, Edited by Merete Lie and Knut H. Sørensen, 1996, pp. 121-150.

Michael Jackson, *Principles of Software Design*, Academic Press, Inc., 1975.

Michael Jackson, *System Development*, Prentice-Hall, New Jersey, 1983.

Heinz H. Klein and Michael D. Myers, “A Set of Principles for Conducting and Evaluating Iterpretive Field Studies in Information Systems”, *MIS Quarterly*, MIS Quarterly, Volume 23 Issue 1, 1999, pp. 67-94.

Thomas S. Kuhn, *The Structure of Scientific Revolutions*, Third Edition, The University of Chicago Press, Chicago, 1996.

Claude Lévi-Strauss, *The Savage Mind*, University of Chicago, Chicago, 1966.

Steven Levy, *Hackers: Heroes of the Computer Revolution*, Penguin, New York, 1984.

Joseph Licklider, “Man-Computer Symbiosis”, *IRE Transactions on Human Factors in Electronics*, Volume 1 Issue 1, 1960, pp. 4-11.

Marshall Kirk McKusick, “Twenty Years of Berkeley Unix; From AT&T-owned to Freely Distributable”, *Open Sources: Voices from the Open Source Revolution*, Edited by Chris DiBona, Sam Ockman and Mark Stone, O’Reilly & Associates, Inc, Sebastopol, 1999, pp. 31-46.

Steven E Miller, “Political Implications of Participatory Design”, *PCD’92: Proceedings of the Participatory Design Conference*, Edited by M J Muller, S Kuhn and J.A. Meskill, Computer Professionals for Social Responsibility, Palo Alto, 1992, pp. 93-100.

Eric Monteiro, “Scaling Information Infrastructure: The Case of Next-Generation IP on the Internet”, *The Information Society*, Volume 14 Issue 3, 1998, pp. 229-245.

Eric Monteiro and Vidar Hepsøe, “Infrastructure Strategy Formation: Seize the Day at Statoil”, *From Control to Drift*, Edited by Claudio Ciborra, Oxford University Press, Oxford, 1998, pp. 148-171.

Jae Yun Moon and Lee Sproull, “Essence of Distributed Work: The Case of the Linux Kernel”, *First Monday (http://www.firtsmonday.org)*, Volume 5 Issue 11, 2000.

Enid Mumford, *Job Satisfaction: A Study of Computer Specialists*, Longman, London, 1972.

Peter Naur, “Programming as Theory Building”, *Agile Software Development*, Addison-Wesley, Reading, 2002, pp. 227-240.

Ikujiro Nonaka and Hirotaka Takeuchi, *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*, Oxford University Press, Oxford, 1995.

Ikujiro Nonaka and Hirotaka Takeuchi, “A Theory of the Firm’s Knowledge-Creation Dynamics”, *The Dynamic Firm: The Role of Technology, Strategy, Organization, and Regions*, Edited by Alfred D. Chandler, Peter Hagström, Örjan Sölvell, Oxford University Press, Oxford, 1998, pp. 214-241.

Wanda J. Orlikowski, “Learning from Notes: Organizational Issues in Groupware Implementation”, *CSCW ’92*, 1992, p. 362-369.

Julian E. Orr, *Talking About Machines: An Ethnography of a Modern Job*, ILR Press/Cornell University Press, Ithaca, 1996.

Steve Palmer and Mac Felsing, *A Practical Guide to Feature-driven Development*, Prentice Hall, 2002.

Michael Polanyi, *Personal Knowledge: Towards a Post-Critical Philosophy*, The University of Chicago Press, Chicago, 1958.

Jon Postel, *Transmission Control Protocol (http://www.ietf.org/rfc/rfc793.txt)*, 1981.

Jon Postel and John Reynolds, *ARPA-Internet Protocol Policy (http://www.ietf.org/rfc/rfc902.txt)*, 1984.

Roger S. Pressman, *Software Engineering: A Practitioner’s Approach*, Third edition, European Adaption, Edited by Darrel Ince, McGraw-Hill Book Company Europe, London, 1994.

Edited by Eric S. Raymond, *The Jargon File (http://www.tuxedo.org/jargon/)*.

Eric S. Raymond, *The New Hacker’s Dictionary*, Third edition, The MIT Press, Cambridge, 1998.

Eric S. Raymond, “A Brief History of Hackerdom”, *Open Sources: Voices from the Open Source Revolution*, Edited by Chris DiBona, Sam Ockman and Mark Stone, O’Reilly & Associates, Inc, Sebastopol, 1999, pp. 19-30.

Eric S. Raymond, *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O’Reilly & Associates, Inc, Sebastopol, 1999.

Dennis Ritchie, “The Evolution of the Unix Time-sharing System”, *AT&Bell Laboratories Technical Journal*, Volume 63 Issue 6 Part 2, 1984, pp. 1577-1593.

Daniel Robey and M. Lynne Markus, “Rituals in Information System Design”, *MIS Quarterly*, MIS Quarterly, Volume 8 Issue 8, 1985, pp. 5-13.

Gilbert Ryle, *The Concept of Mind*, Penguin, Harmondsworth, 1949.

Gilbert Ryle, *Dilemmas: The Tarner Lectures*, Cambridge University Press, Cambridge, 1954.

Richard Stallman, *Original announcement of the GNU project (accessed November 13 2002) (http://www.gnu.org/gnu/initial-announcement.html)*, 1983.

Edited by Richard Stallman, 1984, *What is Copyleft? (accessed November 13 2002) (http://www.gnu.org/licenses/licenses.html#WhatIsCopyleft)*.

W. Richard Stevens, *Unix Network Programming Volume 1: Networking APIs: Sockets and XTI*, Second Edition, Prentice Hall, Upper Saddle River, 1998.

Lucy Suchman, *Plans and Situated Actions: The Problem of Human Machine Communication*, Cambridge University Press, Cambridge, 1987.

Linus Torvalds, *Initial Linux announcement (http://groups.google.com/groups?q=linux+torvalds+group:comp.os.minix&start=50&hl=en 8&scoring=d&selm=1991Oct5.054106.4647%40klaava.Helsinki.FI&rnum=60)*, 1992.

Vinod Valloppilli and Josh Cohen, *The Halloween Documents (http://www.opensource.org/halloween/): Confidential Internal Memorandum on Microsoft’s Strategy against Linux and Open Source (accessed November 13 2002)*, 1998.

Joseph Weizenbaum, *Computer Power and Human Reason: From Judgment to Calculation*, Penguin Books, London, 1976.

Robert Young, *Under the Radar: How Red Hat Changed the Software Business—and Took Microsoft by Surprise*, Coriolis, Sandsdale, 1999.

E-mail Messages

Robert Thau, February 28 1995, *Re: Informing NCSA, archive of the list (http://utoastria.org/thesis/new-httpd/1995/march/0001.html)*.

Robert Thau, March 09 1995, *Re: lights, cameras, action (http://utoastria.org/thesis/new-httpd/1995/march/0054.html)*.

Randy Terbush, March 10 1995, *Multi-homed server support (http://utoastria.org/thesis/new-httpd/1995/march/0097.html)*.

Cliff Skolnick, March 12 1995, *Re: BSDI compilation of apache .001 (http://utoastria.org/thesis/new-httpd/1995/march/0134.html)*.

Rob Hartill, March 14 1995, *non-forking again (http://utoastria.org/thesis/new-httpd/1995/march/0182.html)*.

Cliff Skolnick, March 14 1995, *Re: non-forking again (http://utoastria.org/thesis/new-httpd/1995/march/0184.html)*.

Robert Thau, March 14 1995, *Re: non-forking again (http://utoastria.org/thesis/new-httpd/1995/march/00193.html)*.

Rob McCool, March 14 1995, *Re: non-forking again (http://utoastria.org/thesis/new-httpd/1995/march/0231.html)*.

Cliff Skolnick, March 15 1995, *Process (please read) (http://utoastria.org/thesis/new-httpd/1995/march/0247.html)*.

Robert Thau, March 15 1995, *Re: Process (please read) (http://utoastria.org/thesis/new-httpd/1995/march/00265.html)*.

Rob Hartill, March 15 1995, *patch list vote (http://utoastria.org/thesis/new-httpd/1995/march/0274.html)*.

Brandon Long, March 16 1995, *Re: non-forking again (http://utoastria.org/thesis/new-httpd/1995/march/0301.html)*.

Rob Hartill, March 16 1995, *Re: non-forking again (NCSA 1.4 approach) (http://utoastria.org/thesis/new-httpd/1995/march/0330.html)*.

Rob McCool, March 17 1995, *Re: non-forking again (NCSA 1.4 approach) (http://utoastria.org/thesis/new-httpd/1995/march/0247.html)*.

Rob Hartill, March 17 1995, *Re: non-forking again (NCSA 1.4 approach) (http://utoastria.org/thesis/new-httpd/1995/march/0350.html)*.

Rob Hartill, March 19 1995, *apache-0.2 on hyperreal (http://utoastria.org/thesis/new-httpd/1995/march/0372.html)*.

Rob Hartill, March 21 1995, *Re: vote counting. (http://utoastria.org/thesis/new-httpd/1995/march/0449.html)*.

Randy Terbush, April 01 1995, *Multi-homed (http://utoastria.org/thesis/new-httpd/1995/april/0004.html)*.

Cliff Skolnick, April 01 1995, *Re: Multi-homed (http://utoastria.org/thesis/new-httpd/1995/april/0009.html)*.

Randy Terbush, April 06 1995, *Multi-homed support (http://utoastria.org/thesis/new-httpd/1995/april/0139.html)*.

Brian Behlendorf, April 07 1995, *XBITHACK busted (http://utoastria.org/thesis/new-httpd/1995/april/0166.html)*.

Rob Hartill, April 08 1995, *votes (http://utoastria.org/thesis/new-httpd/1995/april/0229.html)*.

Randy Terbush, April 08 1995, *Re: votes (http://utoastria.org/thesis/new-httpd/1995/april/0230.html)*.

Brian Behlendorf, April 08 1995, *Re: votes (http://utoastria.org/thesis/new-httpd/1995/april/0240.html)*.

David Robinson, April 09 1995, *Re: Virtual Hosts patch (http://utoastria.org/thesis/new-httpd/1995/april/0254.html)*.

Cliff Skolnick, April 09 1995, *Re: Virtual Hosts patch (http://utoastria.org/thesis/new-httpd/1995/april/0256.html)*.

Robert Thau, April 09 1995, *Re: Virtual Hosts patch (http://utoastria.org/thesis/new-httpd/1995/april/0259.html)*.

Rob Hartill, April 09 1995, *NCSA non-forking model doesn't fix the problem (http://utoastria.org/thesis/new-httpd/1995/april/0270.html)*.

Robert Thau, April 12 1995, *Issues for a beta release... (http://utoastria.org/thesis/new-httpd/1995/april/0347.html)*.

Brian Behlendorf, April 12 1995, *Re: Issues for a beta release... (http://utoastria.org/thesis/new-httpd/1995/april/0350.html)*.

Brandon Long, April 12 1995, *Re: Issues for a beta release... (http://utoastria.org/thesis/new-httpd/1995/april/0370.html)*.

Beth Frank, April 12 1995, *Re: Issues for a beta release... (NCSA PLEASE READ) (http://utoastria.org/thesis/new-httpd/1995/april/0409.html)*.

Cliff Skolnick, April 26 1995, *virtual hosts again (http://utoastria.org/thesis/new-httpd/1995/april/0902.html)*.

Andrew Wilson, April 26 1995, *Re: virtual hosts again (http://utoastria.org/thesis/new-httpd/1995/april/0903.html)*.

Robert Thau, May 05 1995, *Arena bugfix still needed (http://utoastria.org/thesis/new-httpd/1995/may/0110.html)*.

Rob Hartill, May 11 1995, *Re: 0.6.2 must go (http://utoastria.org/thesis/new-httpd/1995/may/0196.html)*.

Rob Hartill, May 29 1995, *for_apache_0.7 (http://utoastria.org/thesis/new-httpd/1995/may/0362.html)*.

Brian Behlendorf, June 06 1995, *0.7.1 compile problem (http://utoastria.org/thesis/new-httpd/1995/june/0029.html)*.

Mark Cox, June 07 1995, *0.7.1 problems (http://utoastria.org/thesis/new-httpd/1995/june/0042.html)*.

Ryan Watkins, June 08 1995, *dead children left and right (http://utoastria.org/thesis/new-httpd/1995/june/0070.html)*.

Robert Thau, June 13 1995, *My garage poject... (http://utoastria.org/thesis/new-httpd/1995/june/0137.html)*.

Rob Hartill, June 16 1995, *update (http://utoastria.org/thesis/new-httpd/1995/june/0147.html)*.

Rob Hartill, June 17 1995, *update ii (http://utoastria.org/thesis/new-httpd/1995/june/0152.html)*.

Brian Behlendorf, June 20 1995, *leaking fd’s (http://utoastria.org/thesis/new-httpd/1995/june/0180.html)*.

David Robinson, June 26 1995, *Apache 0.7: too many processes (http://utoastria.org/thesis/new-httpd/1995/june/0216.html)*.

Randy Terbush, June 29 1995, *Shambhala (http://utoastria.org/thesis/new-httpd/1995/june/0244.html)*.

Randy Terbush, June 30 1995, *Re: Shambhala (http://utoastria.org/thesis/new-httpd/1995/june/0246.html)*.

Brian Behlendorf, June 30 1995, *Re: Shambhala (http://utoastria.org/thesis/new-httpd/1995/june/0251.html)*.

Rob Hartill, June 30 1995, *Re: Shambhala (http://utoastria.org/thesis/new-httpd/1995/june/0247.html)*.

Rob Hartill, June 30 1995, *Re: Shambhala (http://utoastria.org/thesis/new-httpd/1995/june/0250.html)*.

Rob Hartill, August 28 1995, *what bugs are there to squash (http://utoastria.org/thesis/new-httpd/1995/august/0873.html)*.

Robert Thau, August 31 1995, *Re: Vote change... (http://utoastria.org/thesis/new-httpd/1995/august/1012.html)*.

Rob Hartill, October 02 1995, *2:0 (http://utoastria.org/thesis/new-httpd/1995/oct/0020.html)*.

Rob Hartill, October 05 1995, *1.0 timetable still up (http://utoastria.org/thesis/new-httpd/1995/oct/0073.html)*.

Roy Fielding, October 15 1995, *Re: Operating System Independence (http://utoastria.org/thesis/new-httpd/1995/oct/0271.html)*.

Ben Laurie, October 16 1995, *SSL/Apache (http://utoastria.org/thesis/new-httpd/1995/oct/0271.html)*.

Robert Thau, October 17 1995, *Apache 0.8.15 on hyeperreal... (http://utoastria.org/thesis/new-httpd/1995/oct/0302.html)*.

Rob Hartill, October 18 1995, *more preparation for 1.0 (http://utoastria.org/thesis/new-httpd/1995/oct/0361.html)*.

Robert Thau, October 24 1995, *Re: 1.oh (http://utoastria.org/thesis/new-httpd/1995/oct/0493.html)*.

Rob Hartill, October 26 1995, *FYI.. (fwd) (http://utoastria.org/thesis/new-httpd/1995/oct/0544.html)*.

Robert Thau, November 05 1995, *New 0.8.16 on hyperreal (http://utoastria.org/thesis/new-httpd/1995/nov/0070.html)*.

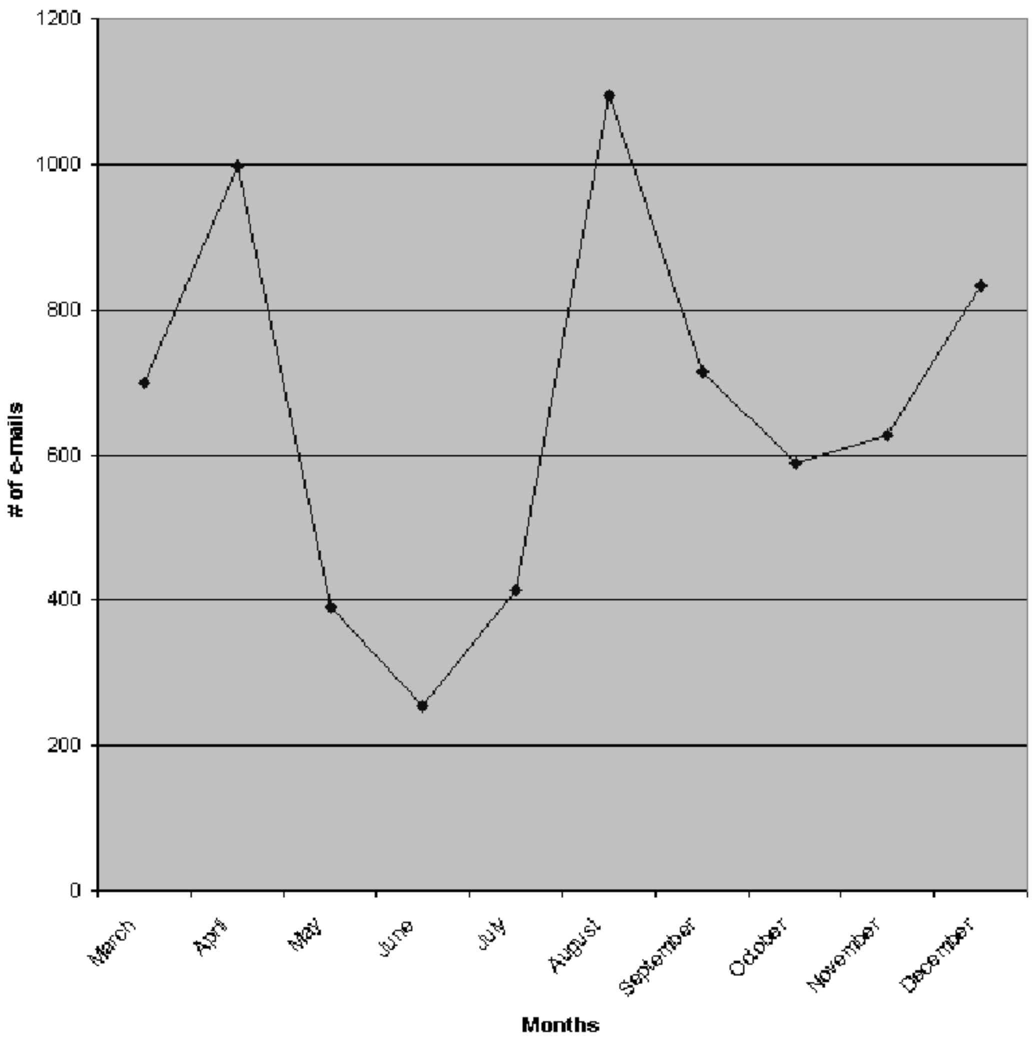
Rob Hartill, November 14 1995, *1.0 (http://utoastria.org/thesis/new-httpd/1995/oct/0153.html)*.

Randy Terbush, November 25 1995, *1.0 Binaries (http://utoastria.org/thesis/new-httpd/1995/oct/0439.html)*.

Appendix A. Statistical Analysis of new-httpd

Traffic analysis 1995

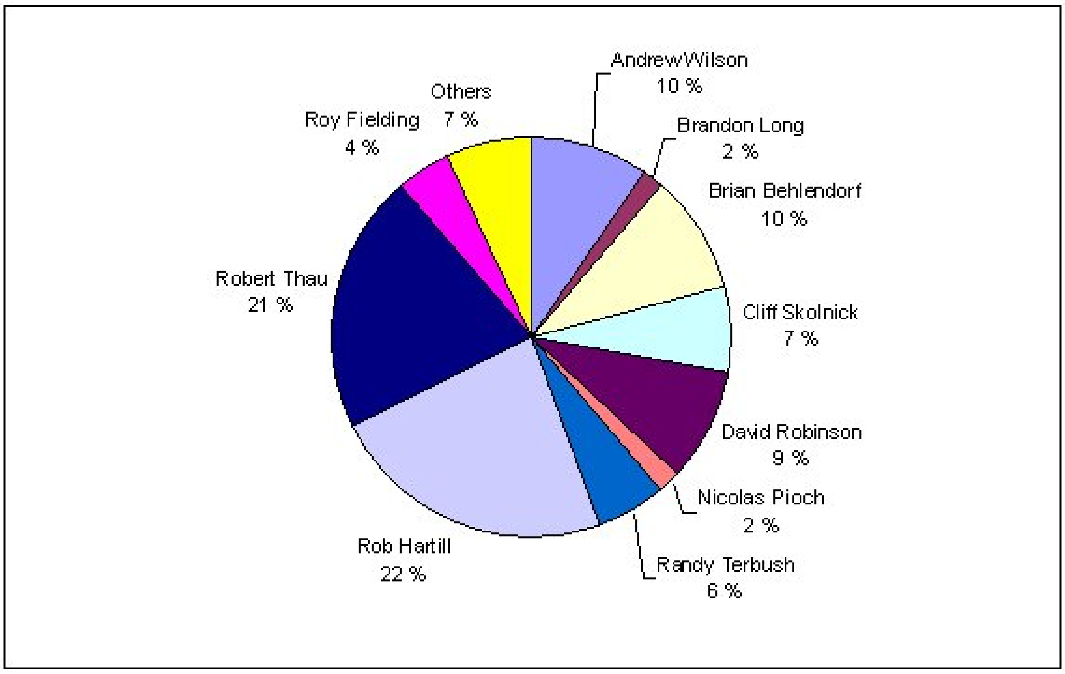
The following graph shows the total number of e-mails sent across the new-httpd mailing list on a monthly basis for the duration of 1995. The reason why January and February isn't included in the graph, is that the mailing list archives start on March 1995.



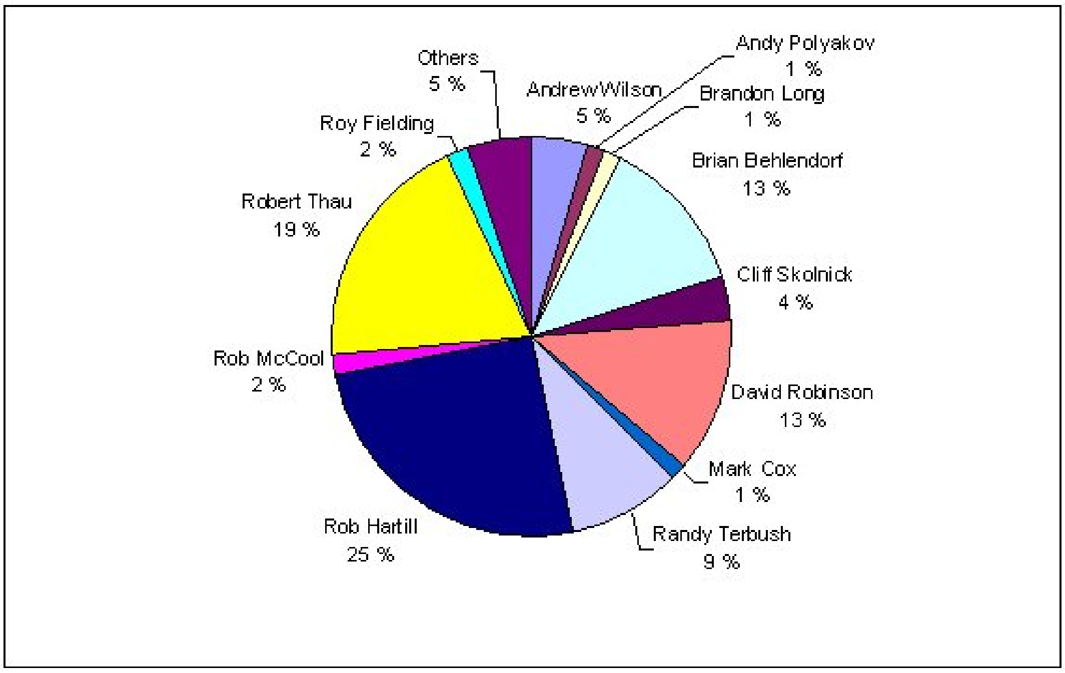
Distribution of e-mail contribution to new-httpd

The following figures shows how many e-mails the individual developers have sent to the new-httpd mailing list over the course of a single month. Those who have contributed with 10 e-mails or less, are collected in the *others* category.

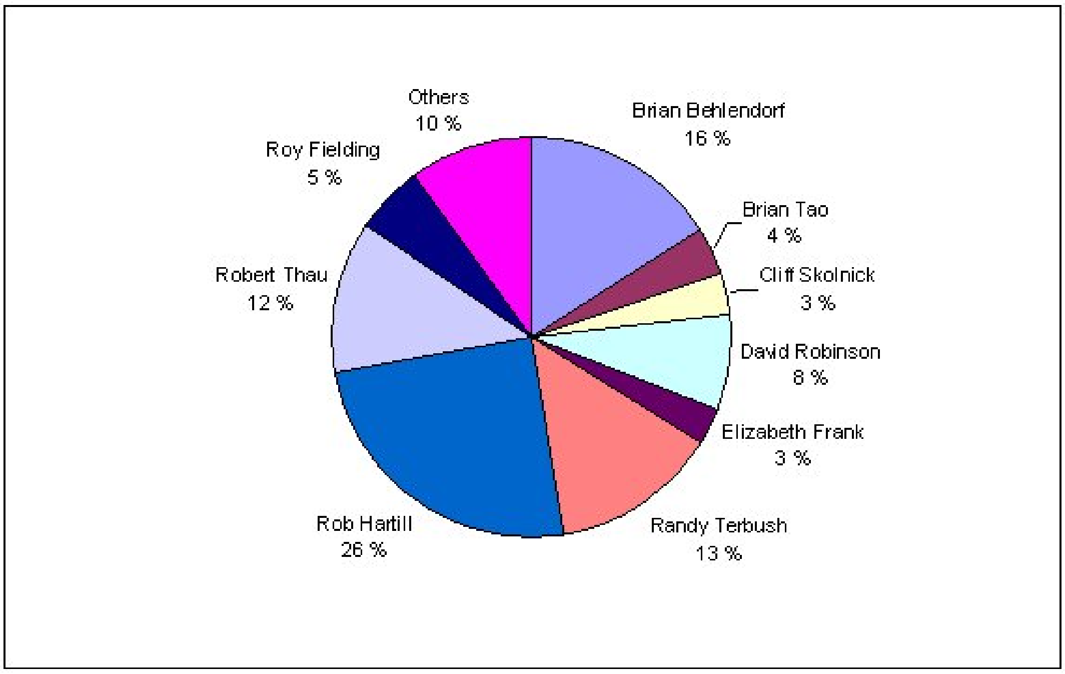
March



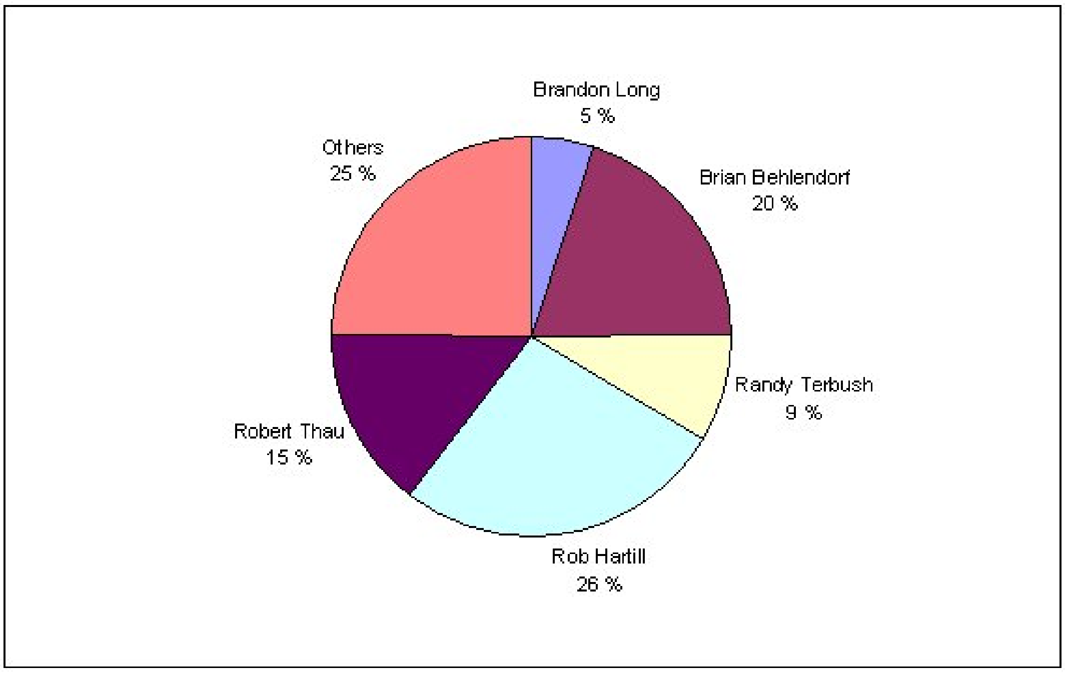
April


Others
5 %
Andrew Wilson
5 %
Andy Polyakov
1 %
Brandon Long
1 %
Brian Behlendorf
13 %
Cliff Skolnick
4 %
David Robinson
13 %
Mark Cox
1 %
Randy Terbush
9 %
Rob Hartill
25 %
Rob McCool
2 %
Robert Thau
19 %
Roy Fielding
2 %

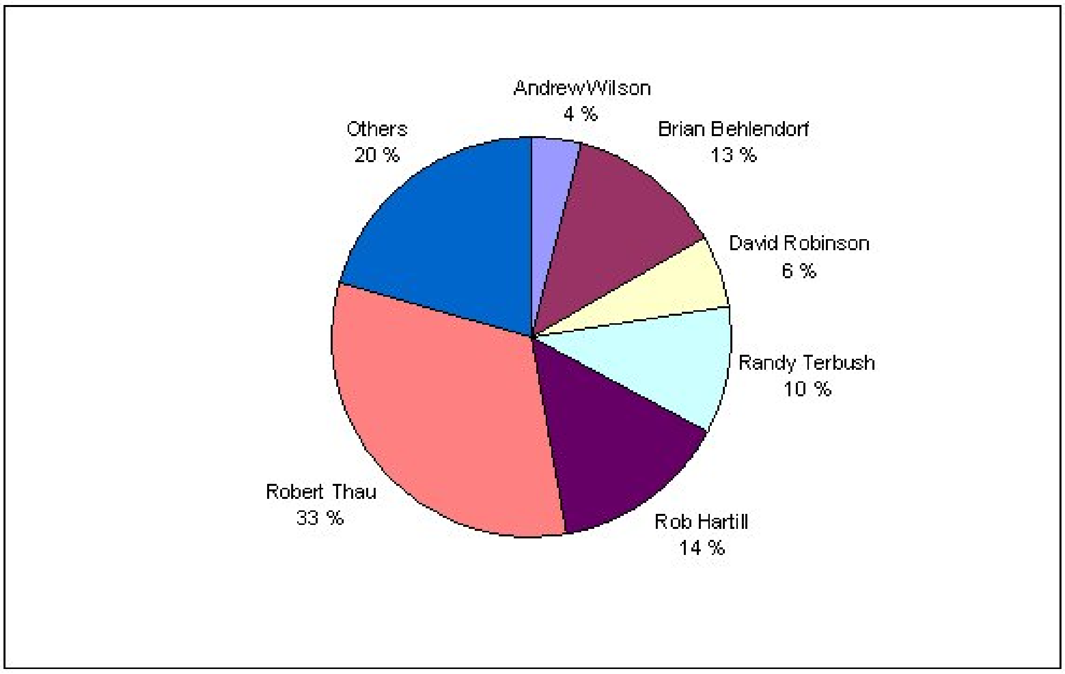
May


Others
10 %
Brian Behlendorf
16 %
Brian Tao
4 %
Cliff Skolnick
3 %
David Robinson
8 %
Elizabeth Frank
3 %
Randy Terbush
13 %
Rob Hartill
26 %
Robert Thau
12 %
Roy Fielding
5 %

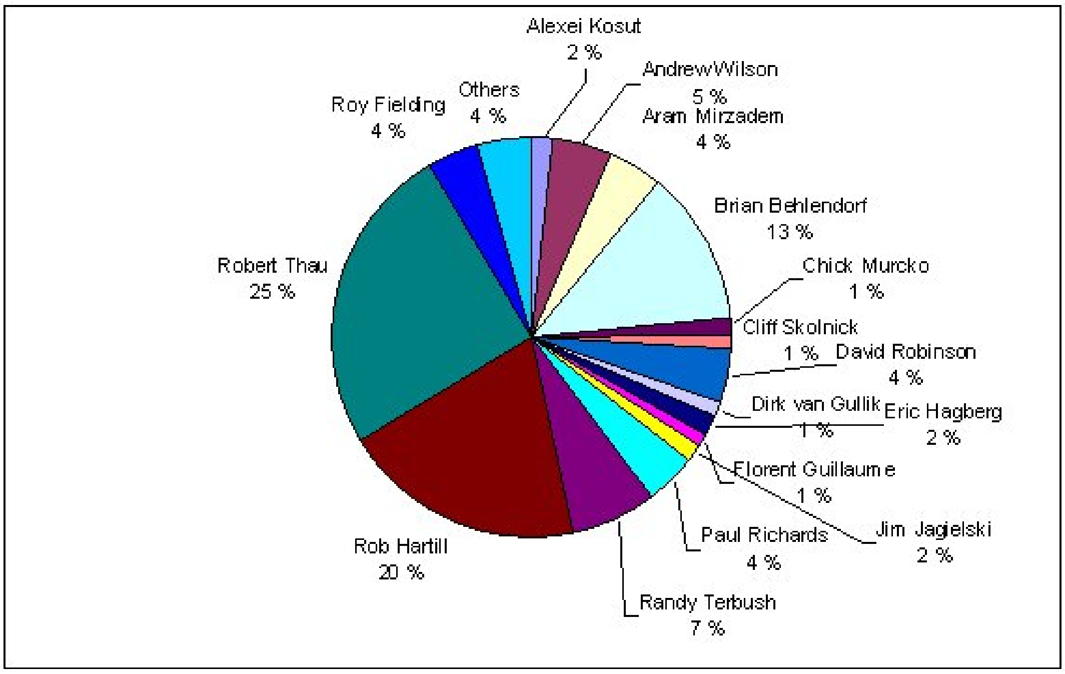
June


Brandon Long
5 %
Brian Behlendorf
20 %
Others
25 %
Randy Terbush
9 %
Robert Thau
15 %
Rob Hartill
26 %

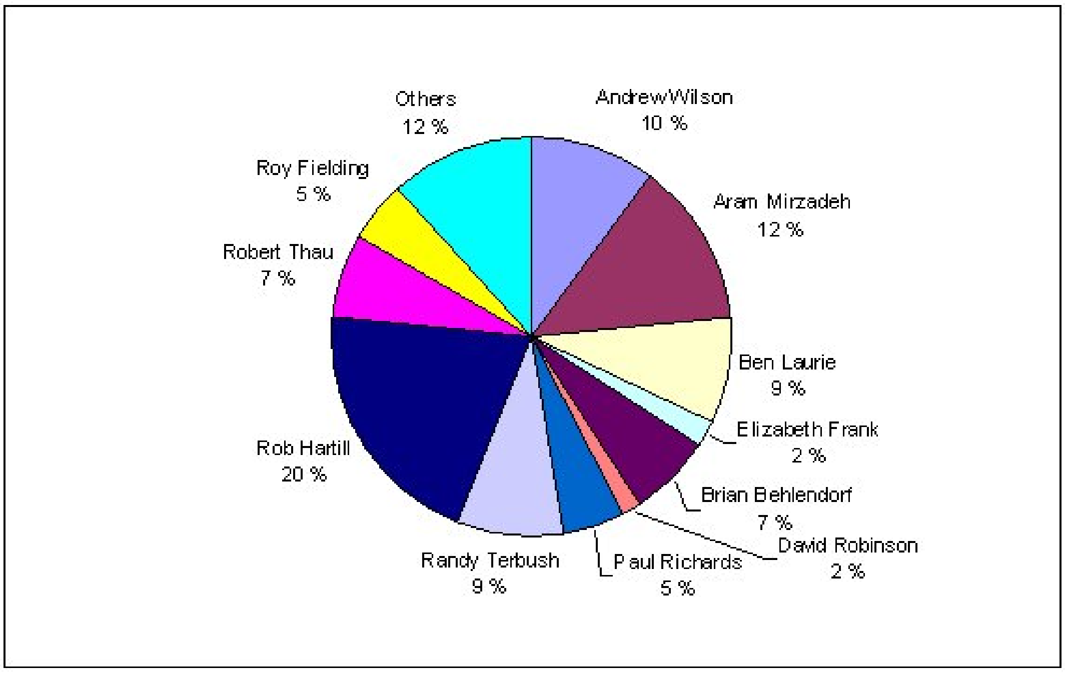
July


AndrewWilson
4 %
Others
20 %
Brian Behlendorf
13 %
David Robinson
6 %
Randy Terbush
10 %
Robert Thau
33 %
Rob Hartill
14 %

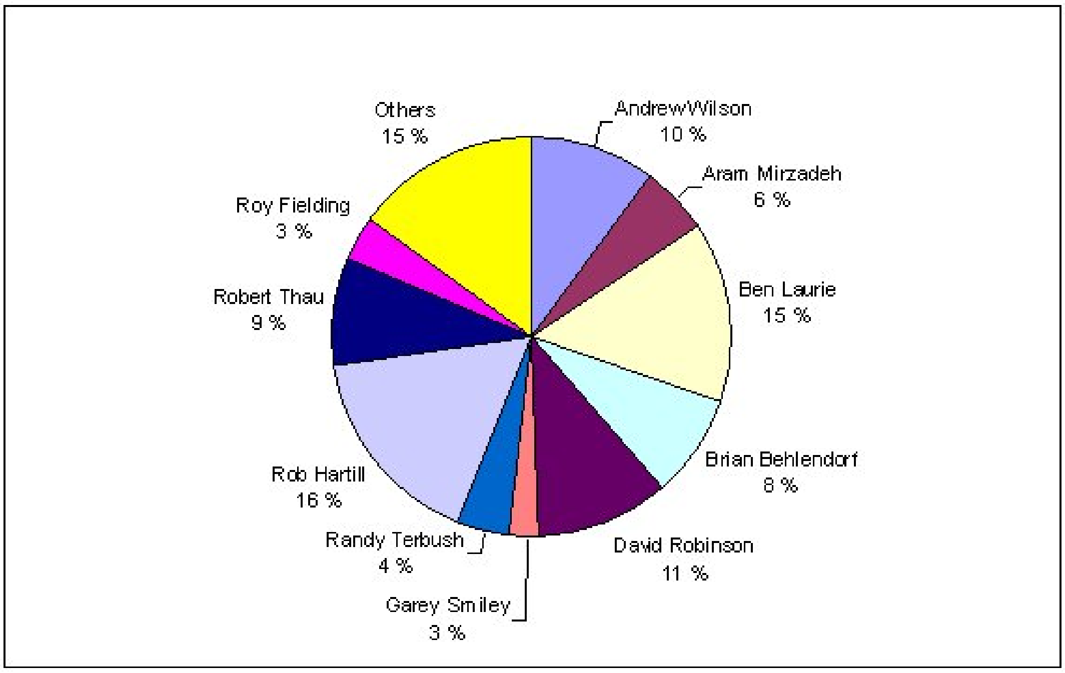
August



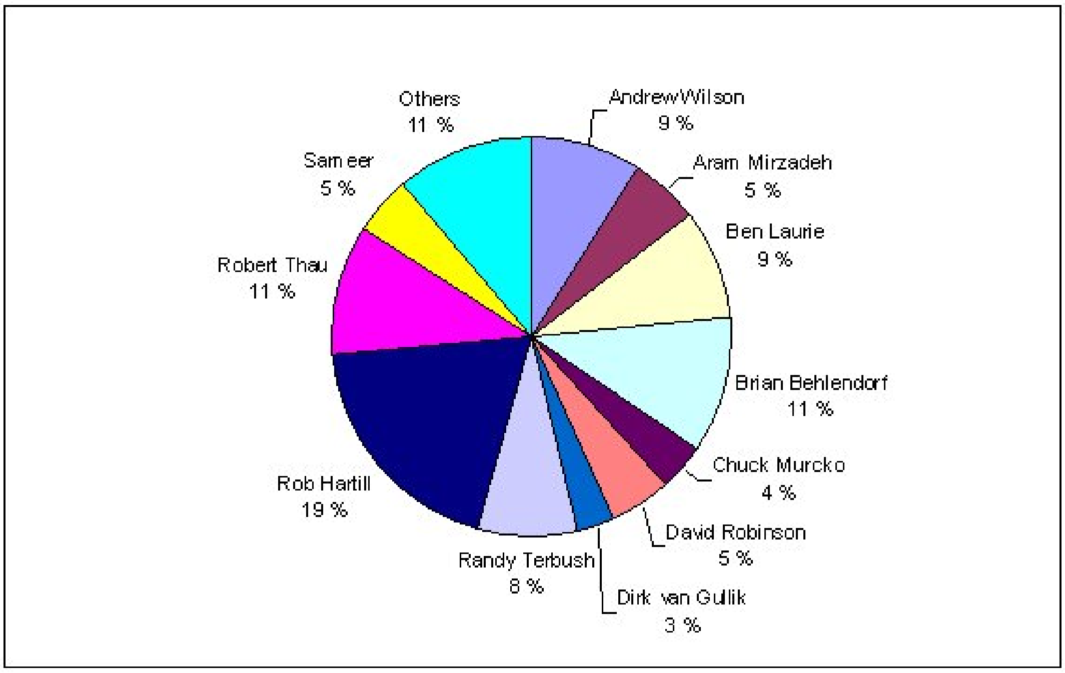
September



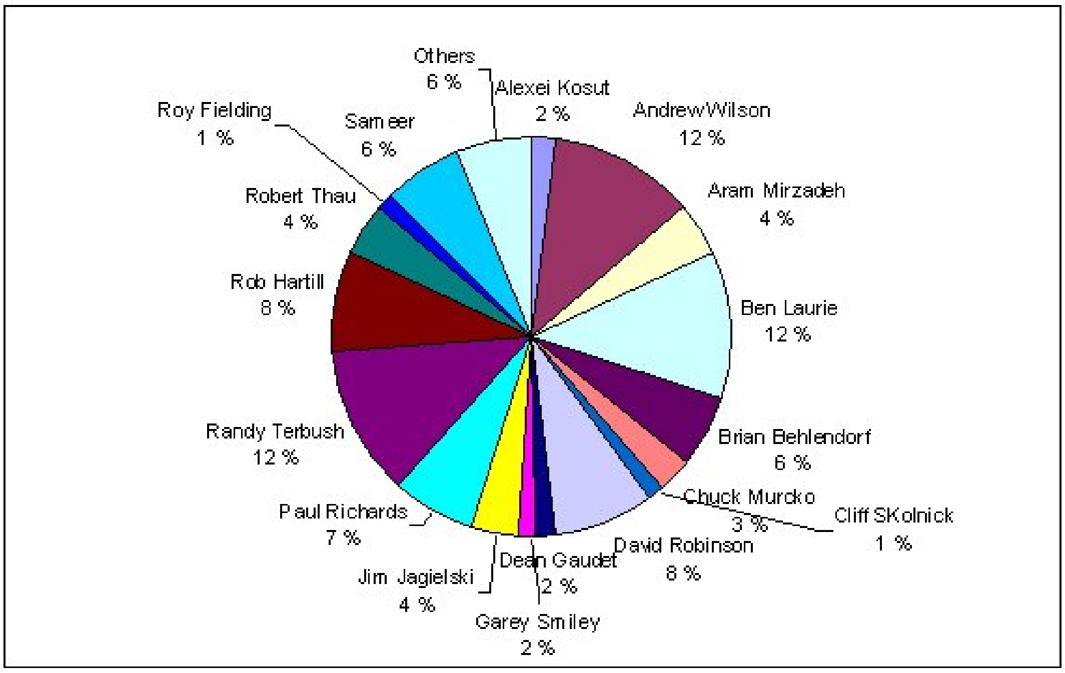
October



November



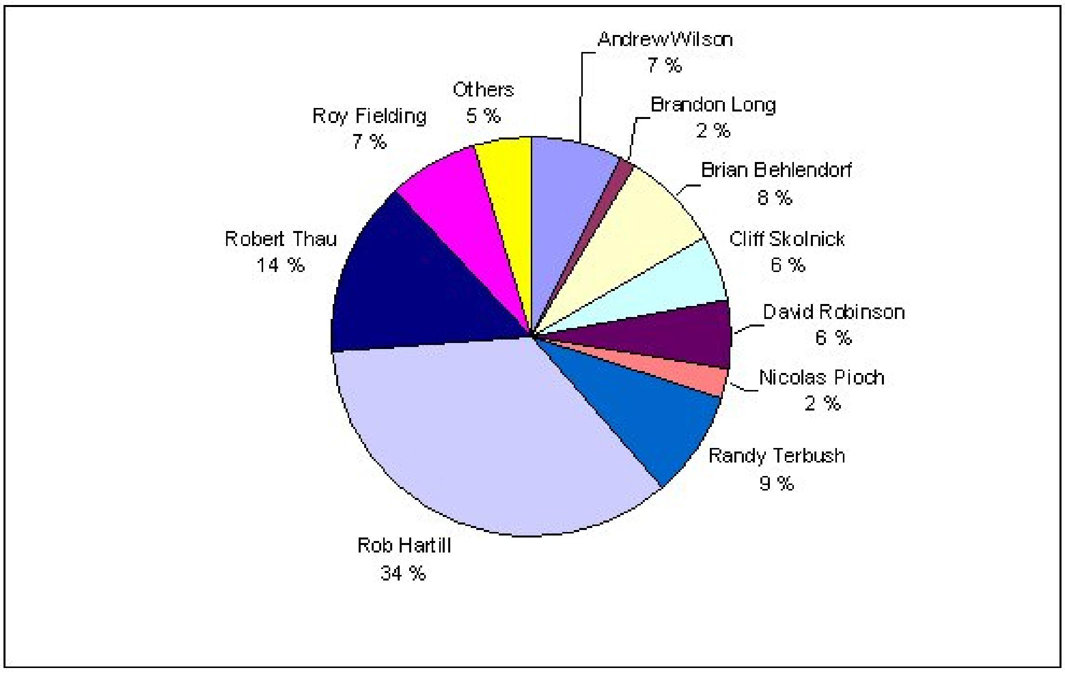
December



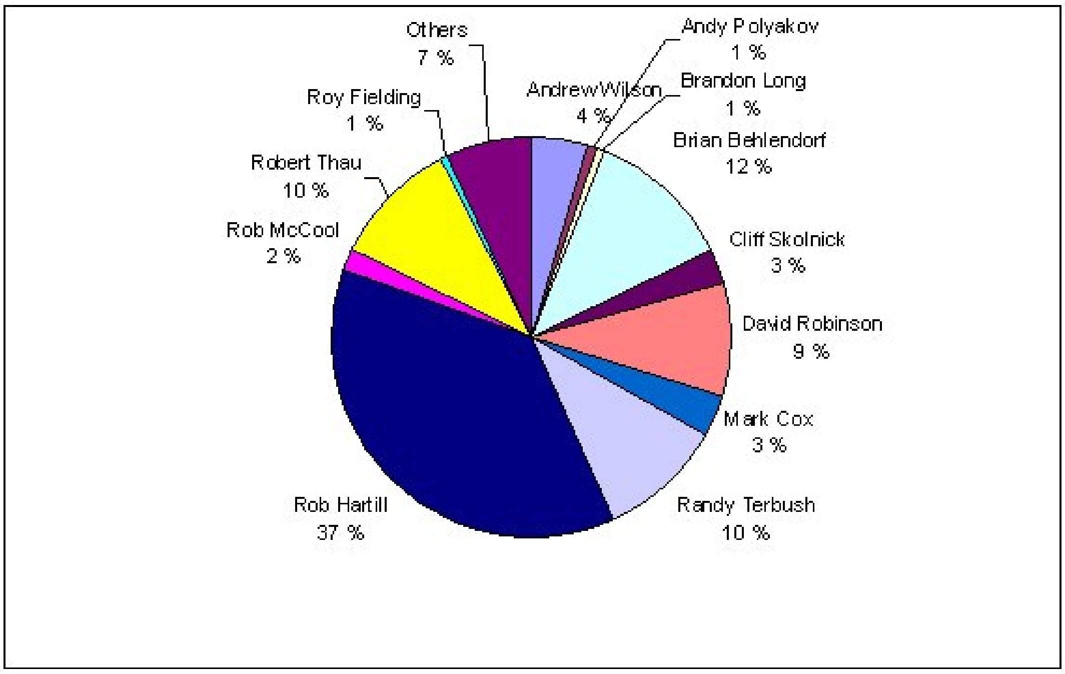
Threads started

The following figures shows how many threads of discussion the individual developers have started to the new-httpd mailing list over the course of a single month. Those who have contributed with 10 e-mails or less, are collected in the *others* category.

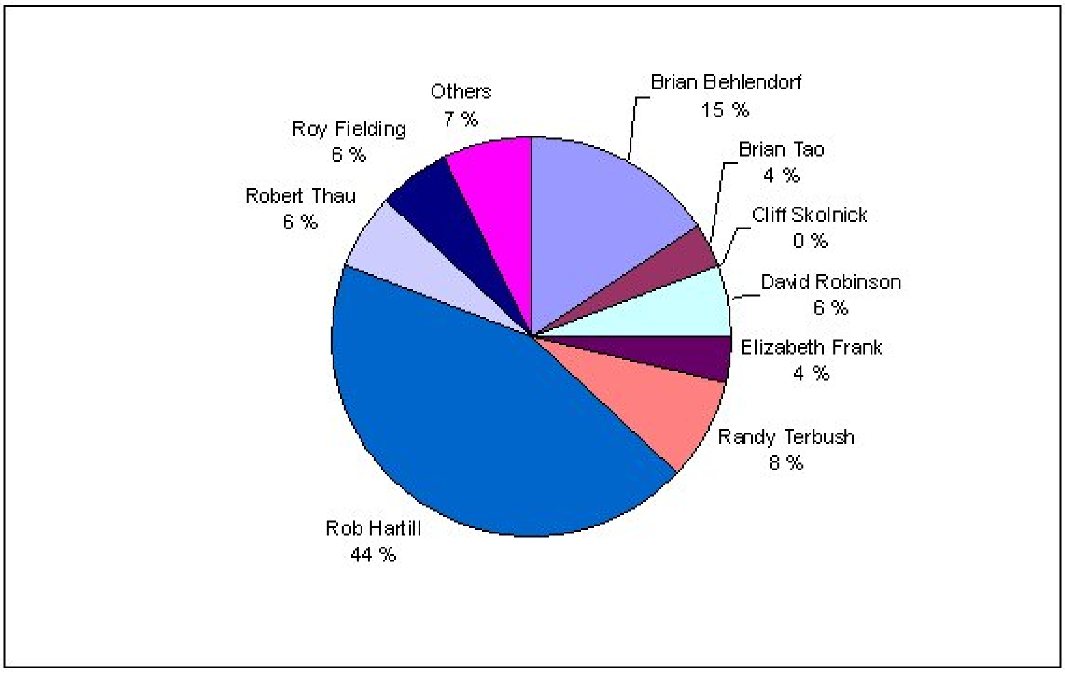
March


AndrewWilson 7 %
Brandon Long 2 %
Brian Behlendorf 8 %
Cliff Skolnick 6 %
David Robinson 6 %
Nicolas Pioch 2 %
Randy Terbush 9 %
Rob Hartill 34 %
Robert Thau 14 %
Roy Fielding 7 %
Others 5 %

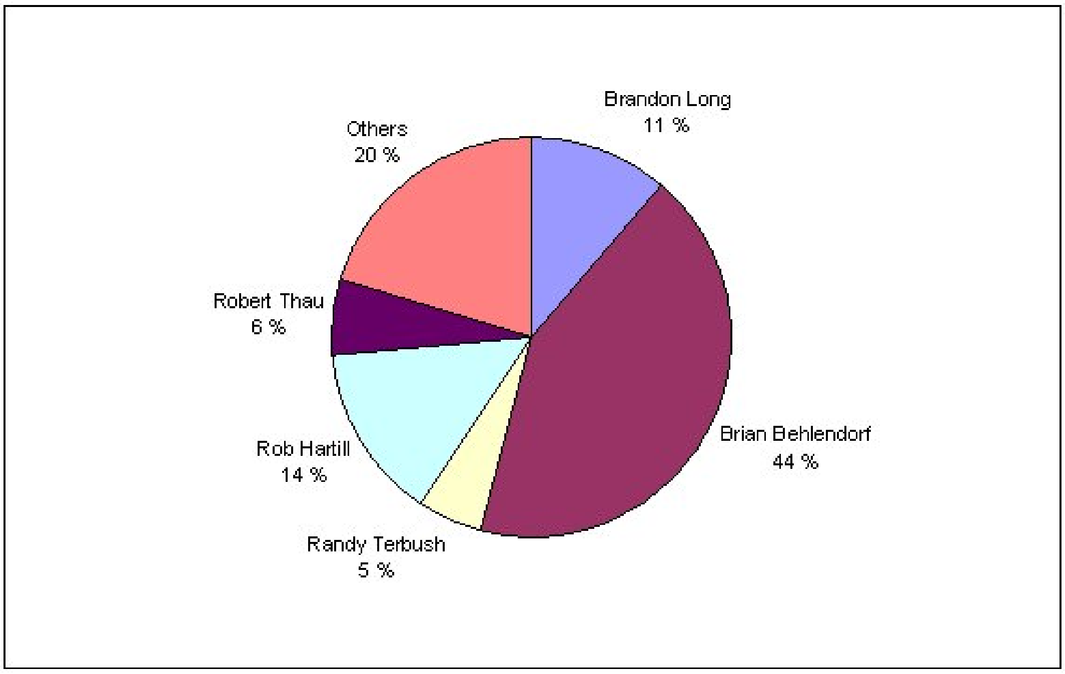
April


Others 7 %
Andy Polyakov 1 %
AndrewWilson 4 %
Brandon Long 1 %
Brian Behlendorf 12 %
Cliff Skolnick 3 %
David Robinson 9 %
Mark Cox 3 %
Randy Terbush 10 %
Rob Hartill 37 %
Rob McCool 2 %
Robert Thau 10 %
Roy Fielding 1 %

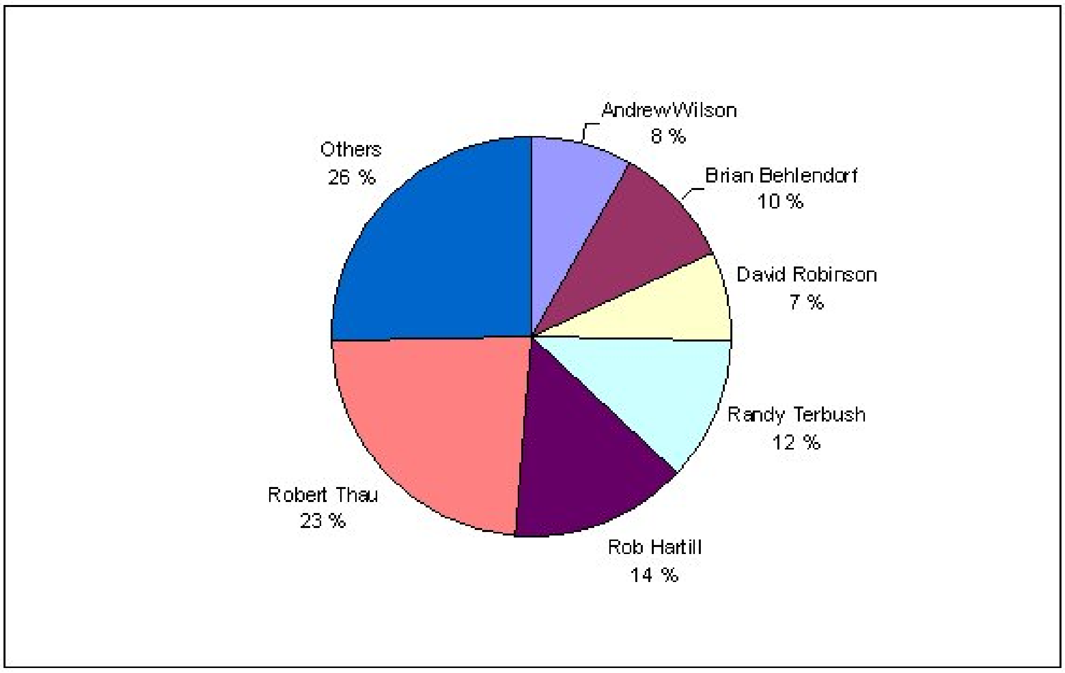
May


Others
7 %
Brian Behlendorf
15 %
Roy Fielding
6 %
Brian Tao
4 %
Robert Thau
6 %
Cliff Skolnick
0 %
David Robinson
6 %
Elizabeth Frank
4 %
Randy Terbush
8 %
Rob Hartill
44 %

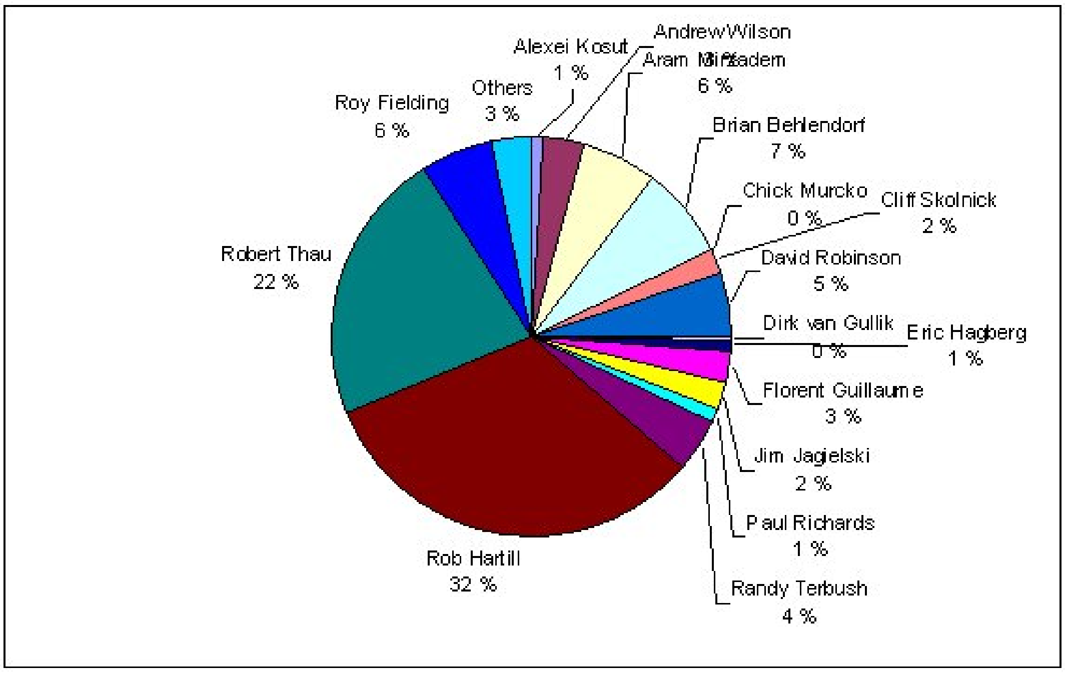
June


Brandon Long
11 %
Others
20 %
Robert Thau
6 %
Brian Behlendorf
44 %
Rob Hartill
14 %
Randy Terbush
5 %

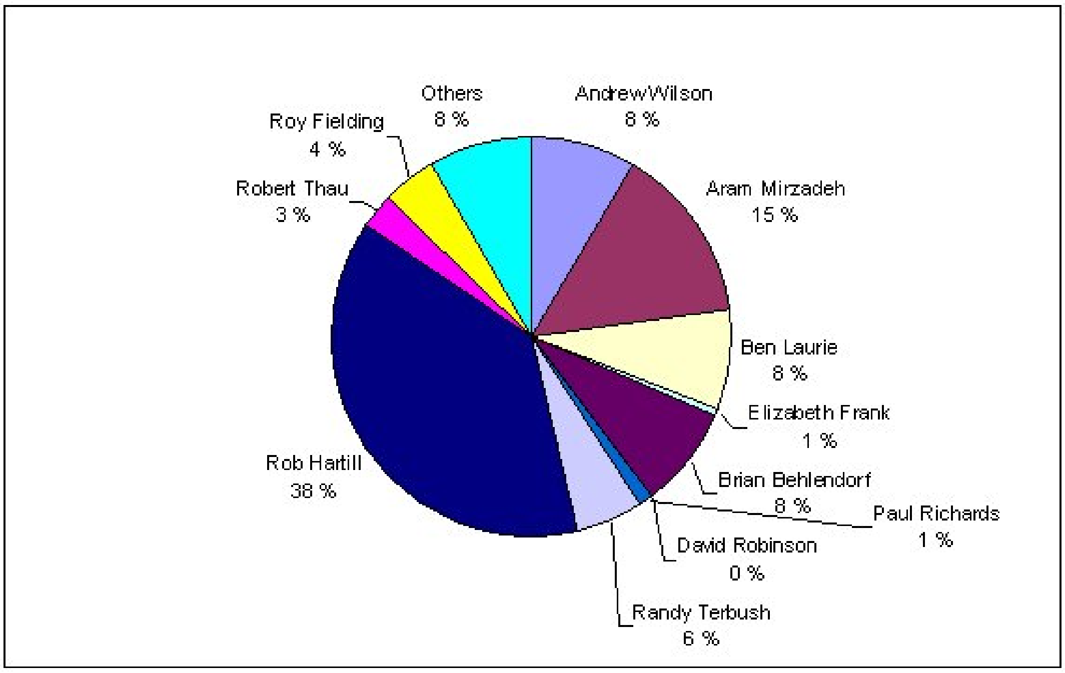
July


Andrew Wilson
8 %
Brian Behlendorf
10 %
David Robinson
7 %
Randy Terbush
12 %
Rob Hartill
14 %
Robert Thau
23 %
Others
26 %

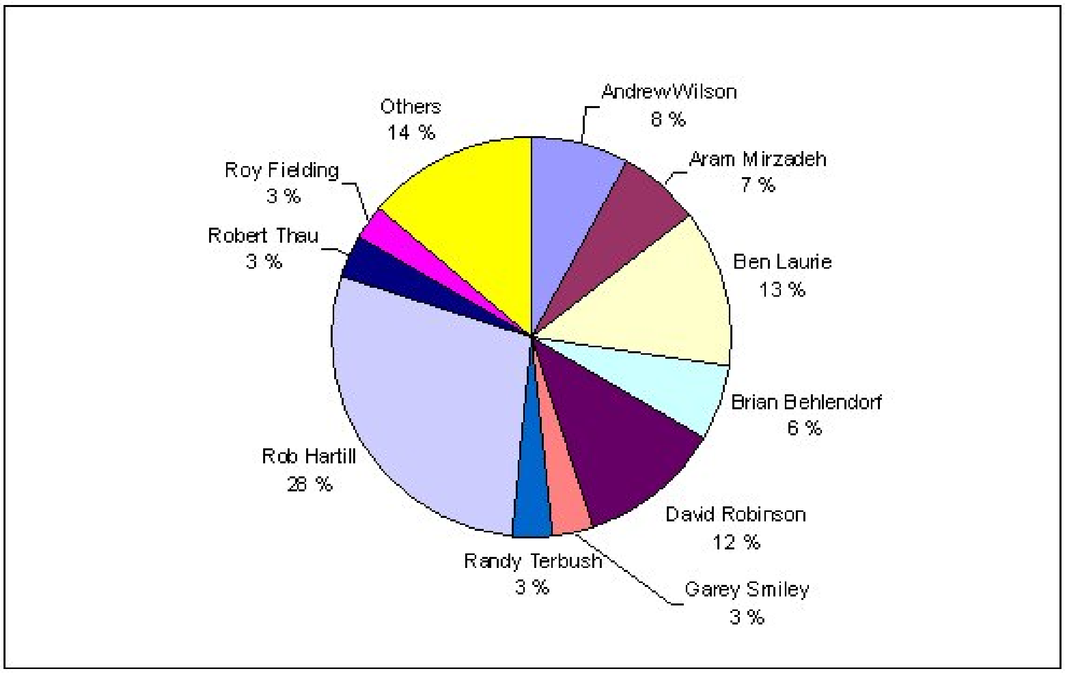
August


Alexei Kosut
1 %
Andrew Wilson
3 %
Aram Mirzadeh
6 %
Brian Behlendorf
7 %
Chick Murcko
0 %
Cliff Skolnick
2 %
David Robinson
5 %
Dirk van Gullik
0 %
Eric Hagberg
1 %
Florent Guillaume
3 %
Jim Jagielski
2 %
Paul Richards
1 %
Randy Terbush
4 %
Rob Hartill
32 %
Robert Thau
22 %
Roy Fielding
6 %
Others
3 %

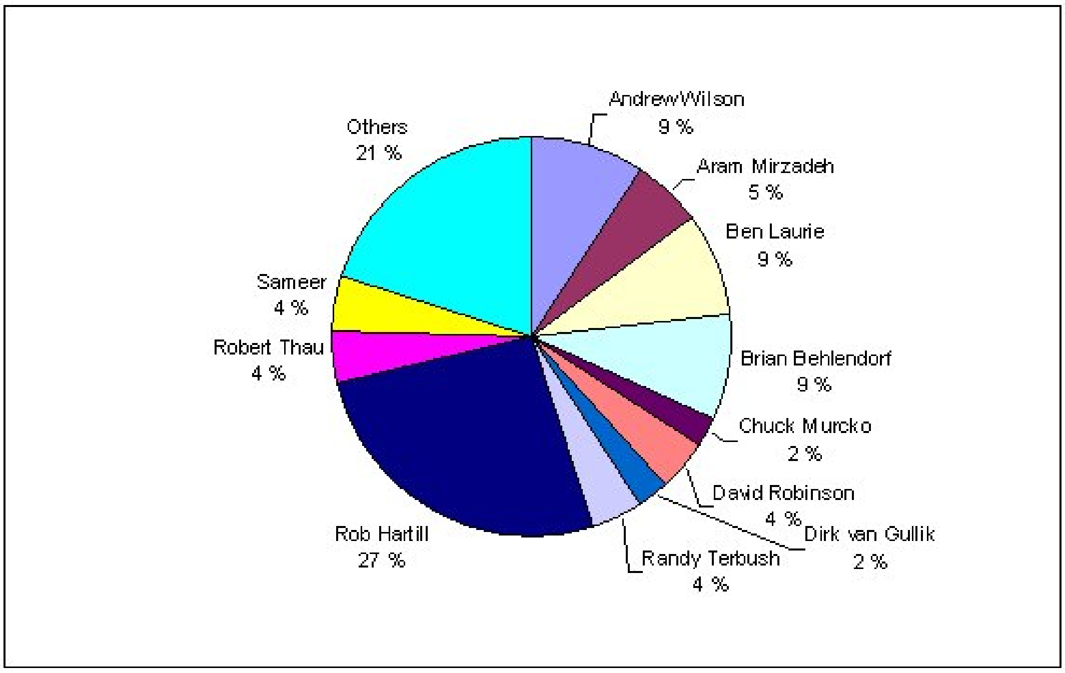
September



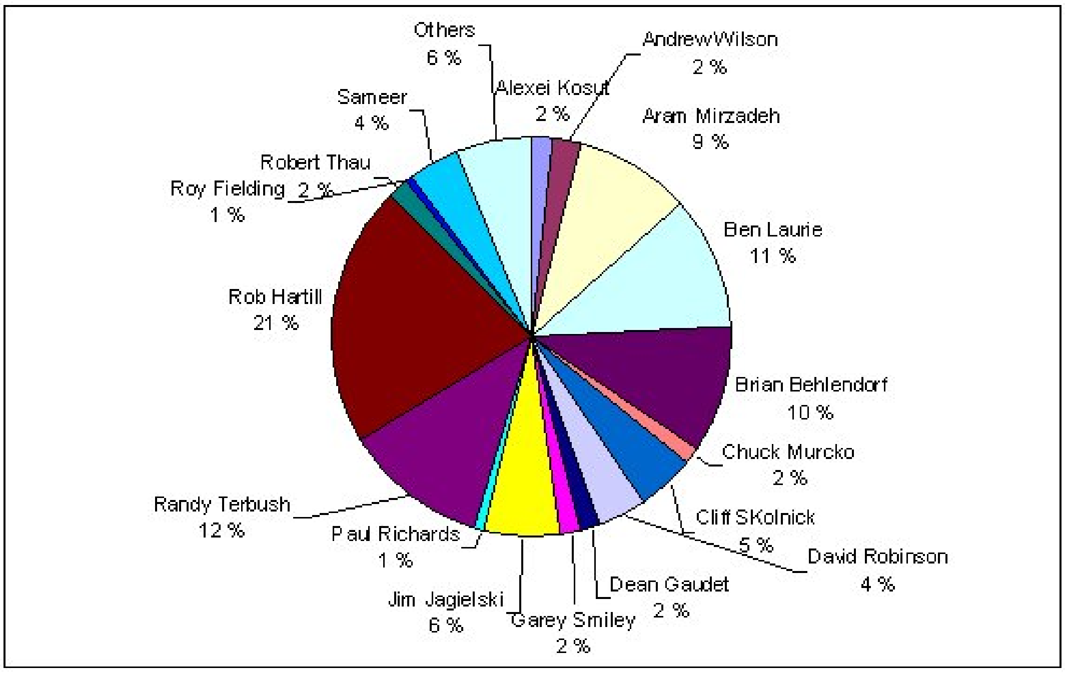
October



November



December



Colophon

This thesis was written on a dual Intel Celeron computer running Gentoo GNU/Linux `1.2` with a `2.4.18` kernel. It was written with GNU Emacs `21.2.1` using Lennart Stafflin's PSGML `1.2.4` major mode for editing SGML and XML documents. It was written in XML using the DocBook `4.0` DTD. Figures were created with Alexander Larsson's dia `0.90`. Spell checking was handled with ispell `3.2.06` US English dictionary. Statistic analysis and charts were done in gnumeric `1.0.8`. Revision control was handled with RCS, the GNU Revision Control System, version `5.7`.

Transformation from XML to intermediary TeX and DVI formats was processed with James Clark's OpenJade `1.3.1` DSSSL implementation and Sebastian Rahtz' JadeTeX `3.12` TeX macros for OpenJade using Noel Walsh's modular DocBook DSSSL stylesheets version `1.64`. Printable versions of the document were converted from DVI to PDF `1.2` by Ghostscript `7.05.5`. The entire transformation toolchain was interfaced through Cees deGroot's SGMLtools-lite `3.0.3` package. Local customizations were made to the default JadeTeX (http://utoastria.org/thesis/jadetex.cfg) and DSSSL (http://utoastria.org/thesis/thesis.dsl) settings.

TimesNewRoman is used for the body copy, Arial is used to create the headings, and literal text is printed in Courier New.