

SvenskMud and Open Source

SvenskMud is – among several other things – most definitely a computer program. Seen as a traditional computer program, or as a traditional software systems development project, SvenskMud ought to be facing a number of insurmountable challenges that, according to all traditional wisdom in the field of software engineering, “should” defeat it, or rather, should *have* defeated it a long time ago. Some of the more serious challenges are:

- SvenskMud is a huge program with millions of lines of code.
- SvenskMud is complex, and internal chains of dependencies in SvenskMud are complex.
- SvenskMud has had more than one hundred different developers come and go as they please over the course of its nine-year long existence.
- Some of the developers have been utter novices at programming.
- System structure in a computer system deteriorates over time as changes are made in the program. SvenskMud did not even have a well-defined structure to begin with.
- SvenskMud, and indeed the original LP-mud source code, is not very well documented, i.e. the rationale for different design decisions are not represented anywhere.
- There is no stated single goal that guides the developments of SvenskMud in an unambiguous way.
- SvenskMud has no project leader who oversees and manages the development of the software in a traditional sense.
- There are no specific plans for how SvenskMud should develop in the future. Or, alternatively, there are as many plans for how SvenskMud should develop in the future as there are magicians developing SvenskMud at this very moment.

Despite these challenges that at first sight seem to be insurmountable, SvenskMud continues to exist, evolve and thrive. Why? How? How are recurrent and continuous challenges, like the software engineering¹ challenges enumerated above, solved in the case of SvenskMud?

To analyze these questions, this chapter starts with a summary of the past 50 years of developing software in the computer industry and a description of traditional software systems development methods. These methods are then contrasted with software development methods of so-called open source software, which seem to go counter to many of the principles behind the traditional methods in the software industry. There are many similarities between how software is developed in the open source movement and how it is developed in SvenskMud though. These will be explored in detail and some differences

¹ Note again that the field is called “software engineering”. I think the term is a contradiction and therefore inappropriate as a *description*, but it could work if it is regarded as a *goal* for the field (c.f. Shaw 1990).

will also be highlighted. Finally, some characteristics that are particular to SvenskMud and other muds will be developed before the chapter is concluded with a discussion.

Open source software has come to the attention of the public, researchers and the (software) industry during the last two years. The concept is strange and alluring and many wonder how it works and what its implications are, but few have dared to predict those yet. This chapter will deal with some of the principles and ideas behind open source, since there exists fundamental similarities between how open source software and SvenskMud is developed. To clarify further; my interest is in understanding and explaining the social aspects behind how *SvenskMud*, as a computer program, is developed. I am interested in open source only to the extent that it can help me understand SvenskMud better, i.e. this is primarily a chapter about SvenskMud, not about open source.

As to beliefs and motivations of the actors, the individual SvenskMud magician-programmers or the individual open source contributor, the next chapter will go into some more detail. Neither this nor the next chapter will however delve into the (economic) dynamics of open-source software (Raymond 1999, McKelvey 2000), nor what the (economic) implications are for the software industry or for different business models (Behlendorf 1999, Ousterhout 1999, Raymond 1999, Young 1999, McKelvey forthcoming 2000). Other subjects that will not be delved into here is the prehistory of the open source movement, why open source is growing and why this is interesting or important in the first place.

Commercial software development

Early software development and the software crisis

In the early 1950s, in the nascence of computing, everything that had to do with computers was of an experimental nature. The computer hardware was at the focus of interest and computers were used to crunch large numbers of numbers (mathematical calculations) in a simple – but to a human being – time-consuming manner. At the time, ordinary people thought of computers in terms of “thinking machines” or “electronic brains” (Martin 1993).

As computers began to be sold on a commercial basis, new applications were developed and software became more important. The first high-level computer languages like FORTRAN and COBOL were created and software development started to become separated from hardware development. At this time, software was created together with new models of mainframe computers and included in the sale of the computer (hardware). As a rule, software for one computer did not work on other computers (from other computer manufacturers or even later series of models from the same computer manufacturer). The high-level computer languages solved many earlier problems of software development, for a time. The remaining problems were explained by the inexperience of (some) programmers. The problems would disappear as (these) programmers became more experienced.

That did not happen. Since even experienced programmers failed to produce high-quality computer programs on schedule and at cost, the computer industry started to talk about a crisis in software development. The perceived scope of the crisis can be illustrated with Brooks (1975/1995) ominous description, comparing the problems of the software industry to a pre-historic tar pit from which no one escapes:

“No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind’s eye one sees dinosaurs, mammoths, and sabertoothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.

Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems – few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty – any particular paw can be pulled away. But the accumulation of simultaneous and interactive factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem and it is hard to discern the nature of it.”

Brooks, 1975/1995, p.4

Programming expertise was no remedy since human expertise only went so far in the face of exponentially growing complexity of larger and larger projects and larger and larger numbers of programmers. “A fundamental characteristic of many software systems is that they are very large and far beyond the ability of any individual or small group to create or even to understand in detail” (Kraut & Streeter, 1995, p.69). And the problem did not go away, but became worse over time. In the end, something new, something radically different was needed; an understanding of software development as *a process* of solving problems. What were needed were methods for creating order in that process.

The waterfall model

It was from the developments described above that different “waterfall” models of systems development were introduced at the end of the 1960’s and the 1970’s². The waterfall model was the standard systems development model during the 1970’s and 1980’s and it still remains the basic model in many places today.

The waterfall model divides the process of developing computer software into a number of phases, and places each phase in its specific place in a sequential chain of steps. Typically the steps are graphically depicted one after another in a falling fashion left-to-right, the development process flowing forward/down, just like a waterfall (see figure 6.1 below). The waterfall model basically contains phases for

- 1) Analyzing requirements and producing product specifications – specifying *what* the software will do.
- 2) Software design – specifying *how* the software will produce the desired result.
- 3) Coding – production/implementation in a programming language.
- 4) Testing – verifying that the software works and is acceptable to the users.

² Part of this section is based on information from “Programutveckling” [“Software development”] (Olsson, 1980), a textbook that I, myself, used when I studied computer and systems sciences in the end of the 1980’s.

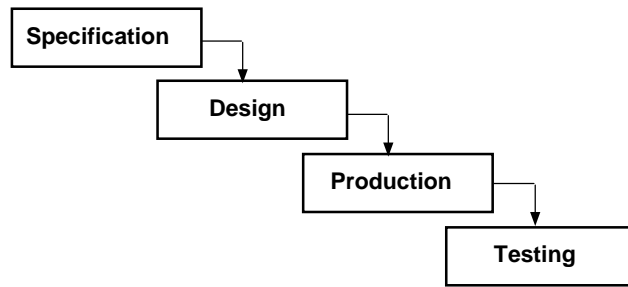


Figure 6.1. The waterfall divides the software process into distinct consecutive steps to make it more manageable.

Different variations of the basic waterfall model can divide the systems development process into different numbers of steps by adding steps (for example documentation and maintenance), or by dividing steps (e.g. phase 1, analyzing requirements into analyzing systems requirements and analyzing software requirements). See figure 6.2 below for an extended version of the waterfall model.

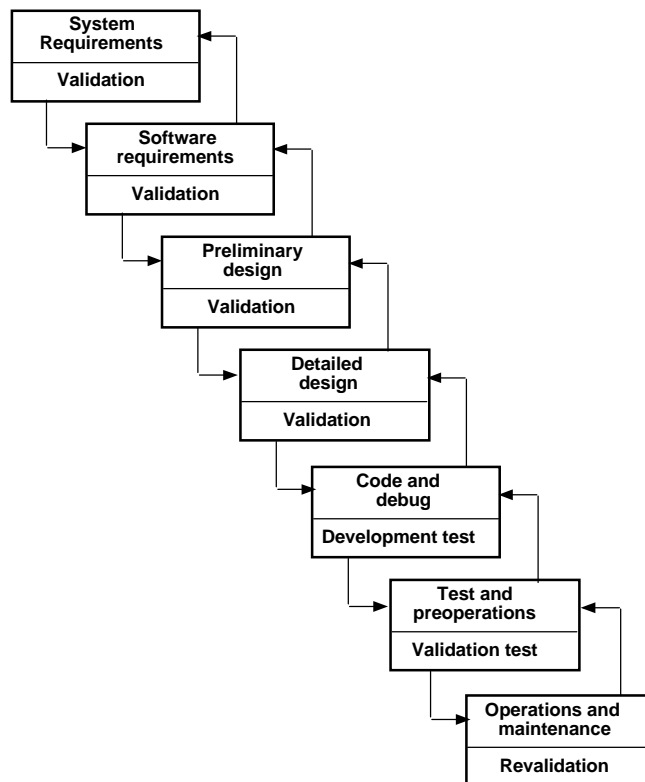


Figure 6.2. The conventional waterfall software development life-cycle model (Boehm, 1976). This version is one of or, perhaps the most influential and most-often reproduced version of the waterfall model.

It does not matter if the waterfall model is minimal or extended, the main principle remains the same, namely to “divide and conquer” the complexity of the software development process. One important characteristic of the waterfall model is that the specifications for the system are frozen at an early stage in the development process. After the system is finished, it is possible to test it and to see if it fulfills the demands as specified. The waterfall model made sense at a time when computers were rare and expensive:

“Because computers were an expensive resource, it made sense in 1970 that access to them should be preceded by careful planning so that time on the machine would be used effectively. The same planning and forethought should accompany the use of any expensive instrument. The waterfall model reflected this aspect of proceeding systematically and deferring implementation. [...] the relative cost to fix errors on large projects increases exponentially depending on the phase in which the error was detected. This economic rationale was the basis for the assumption that successful software was developed by successively achieving subgoals at each phase.”

Agresti, 1986, p.4

“[The waterfall life-cycle model] can be traced to the reductionist mode of inquiry that achieved prominence with scientific work on planning and logistics during and after World War II” (Agresti, 1986, p.2). By introducing a structured way of developing computer programs, the conditions for developing programs of high quality was improved. High quality in a computer program can, in this context, be operationalized as performing well when evaluated in comparison to some difficult-to-attain ideals such as *reliability*, *ease of maintenance*, *ease of modification*, *generality*, *efficiency* and *ease of use*. Hovering above all these criteria are the always-present demands in terms of time and money (cost).

Different ideals may be weighted differently in different projects, e.g. for some programs and for some uses, reliability will be rated higher than ease of modification etc. All ideals can be reached to a higher or lesser degree as the design process in itself constitutes a series of decisions which determine how limited resources should best be used to attain mutually exclusive goals.

Beyond the waterfall model

The waterfall model has proved to be a robust model for developing software, but the model is, in its traditional incarnation, too inflexible for today’s fast-moving world of commercial software. With the waterfall model, after the specifications are frozen (and contracts are written perhaps), the whole point is that no changes should be made in the plans. After specifications are frozen the customer basically gets what he asked for. Unfortunately that might be different from what the purchaser really wanted (which again might differ from the real need). Or, some details were (by necessity) left unspecified and their importance was not realized until later, perhaps after the system was delivered. And as the waterfall model adds structure, it also adds inflexibility.

Because of this the waterfall model has lost grace and often been replaced by a number of more flexible methods for developing software³ during the 1980’s and 1990’s. Proto-

3 The waterfall model has been replaced by different models in different computer science sub-disciplines; by prototyping and iterative models in the area of human-computer interaction, by socio-technical and Scandina-

types and iterative cycles of designing, building and evaluating software-in-progress, in an incremental fashion, is the standard in the software industry today. “The basic idea shared by these approaches is that users’ needs for many types of software are so difficult to understand and that changes in hardware and software technologies are so continuous and rapid, it is unwise to attempt to design a software system completely in advance” (Cusumano & Selby, 1997, p.55).

The context of industrial software development

In contrast to the simple linear developments sketched out above, the process of developing processes for developing software is not a simple movement from bad to good and from good to better, i.e. from the waterfall model to more progressive and better models. To understand the phenomenon one has to understand something about the purposes and in which situation software has been developed. This includes knowledge of social, cultural, historical and institutional factors that has been – and is – affecting the development of software.

Grudin (1990) has written about how, historically, the focus of the design of the interface between humans and computers has shifted from hardware to software and finally to social processes. Somehow simplified, a new frontier has opened up every decade. In the 1950’s the interface between humans and computers was the hardware itself. In the 1960’s, with the advent of high-level computer languages, the interface shifted to the programming task. In the 1970’s, with the advent of time-sharing systems and computer terminals, the interface shifted to keyboards and monitors. In the 1980’s, with the advent of personal computers, the interface shifted to the “dialogue” between the individual user and his or her computer. Finally, in the 1990’s, with the advent of computer networks and systems to support work groups or whole organizations, the interface shifted to social processes. The shift of focus for the interface between humans and computers is consistent with changes in systems development over time and in the *kind* of systems being developed. Again somehow simplified, this focus has also shifted every decade since the inception of modern computers. Starting with government contracts in the 1950’s, the focus has gradually shifted to business systems in the 1960’s, to office systems in the 1970’s, to PC applications in the 1980’s and finally to network applications in the 1990’s.

The *type* of programs developed directly affects *how* the computer programs themselves are developed. With government contracts in the 1950’s, the specific function(s) of a future computer system was first identified and specified and different external companies were invited to bid on the contract to develop the software. With business and office systems, much of the development in large companies were done in-house, by the internal computer department. It has to be kept in mind that until the 1980’s, computers were not owned and used by individuals (consumers), but only by corporations and governmental agencies. At the present, fewer and fewer institutions develop their own software⁴.

With an understanding of these relations, it is easier to understand where the waterfall model comes from; for the original software development projects involving government

vian models (involving participation of the end-users in the development process) in the area of information systems and by spiral models in the area of software engineering.

⁴ Many institutions today rather buy and put together a variety of off-the-shelf products and customize them to fit their operations.

contracts and external bidders, it makes excellent sense to use such a model. In such an environment, the requirements were hammered out over the course of months, perhaps years, before external companies made bids on the contract. However, the waterfall model works best when everything a system will perform can be decided and specified (in detail) in advance of developing the system. It does not work as well when the goal is to develop software to support a non-specific need or to develop innovative software “in search of a market”.

How hackers develop computer programs

At times it can be very difficult to differ between what hackers *do* and what hackers *are* (and believe), as well as to differ between myth, reality and posturing. To a large extent a hacker is a person who is (very) good at programming and who believes in some variant of “the hacker ethic” (Raymond 1996). Despite this, I will for now leave the matter of hacker attitudes and motivations (what hackers *are/believe*) until the next chapter and here concentrate on what hackers *do*, i.e. program computers. The first two out of eight definitions of the word “hacker” in *The new hacker’s dictionary* (Raymond, 1996) states that a 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.”
- “one who programs enthusiastically (even obsessively) [...]”.

A hacker programs computers because a computer constitutes an intellectual challenge; it exists, it is complex and alluring, but it always follows the rules and thus *can* be mastered. In terms of motivation, a hacker differs from most computer professionals who work with computers for a variety of reasons of which fascination might be one, but hardly the most important in day-to-day work (very important is probably the fact that they receive a salary). Consequently, hackers develop programs in a different manner compared to computer professionals.

In the very first pages of Brooks’ seminal book about software engineering, *The mythical man-month* (1975/1995), he clarifies the difference between a *program* and a *programming systems product*. His pointed rhetorical question is: How come not all industrial programming teams have been replaced by dedicated garage duos when we can read newspaper accounts of how two programmers (i.e. hackers) in a garage have built an important program that surpasses the best efforts of large teams? His answer lies in answering the question of *what* is being produced in these different environments.

Going from a program to a programming systems product involves two costly steps. *The first* involves developing the program to a *programming product* that “can be run, tested, repaired, and extended by anybody [and that] is usable in many operating environments, for many sets of data” (ibid., pp.5-6). *The second* involves developing a program to a *programming system*, “a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks” (ibid., p.6). Brooks’ estimate is that each of these two steps increases the effort (i.e. the production costs) with a factor of three and that taking both of these steps – develop-

ing a *program* into a *programming systems product* – increases the effort with a factor of nine.

In commercial software development projects, no matter what the software is actually going to *do*, the underlying goal is always the same – to develop high-quality software with limited resources. “People who develop complex system software [...] point out that developing good software is not so much a matter of writing good individual programs as it is of writing a variety of programs that interact well with each other in a complex system” (Ceruzzi, 19991, p.81). For hackers the issue at stake is different. The goal is “the same” – to develop good software. Good software in an industrial setting differs from the criteria that hackers use to evaluate what is quality and what is not. To a hacker, the emphasis is primarily on solving a difficult problem here and now, not on developing a robust *product* that will work in a variety of circumstances and in a variety of environments.

Hackers can of course work for companies and this can obviously work out better in some situations than others. Stone (1995) vividly describes the clash between managers from the traditional school and hackers at Atari in the beginning of the 1980's. Atari knew that their hackers were laying golden eggs as sales and profits were growing exponentially and they did not want to interfere in any way with the hackers' day-to-day working habits. Still, it was thought that the state of affairs was a temporary phase and the company tended to “recruit project managers from places with good, solid histories of developing *real* software – companies with proven track records and dependable profitability – companies like Lockheed, General Dynamics, Martin Marietta, and McDonnell Douglas” (Stone, p.129). The clash was inevitable.

“Soon Atari had project managers [...] [whose] skills they brought to the game market were acquired by designing missile launchers and tank guidance controls. They saw no particular difference between missile software and interactive games. After all, they said, both were software. [...] The timelines and reliability requirements for military software are frequently created in a pork-barrel milieu. The software has to be extremely reliable, although in practice no software is completely reliable. The projects are frequently immense, and therefore the project managers break the software up into chunks and parcel the writing out to teams. While it is possible to have a general overview of the project, the individual team don't know precisely what the other teams are doing on the level of the code itself. And, as with any large project, any changes anywhere in the system must have a paper trail – revision requests, authorizations, confirmations. These are well-worn and quite serviceable histories of successful large-project management.

Because these things take time and are frequently designed to be expensive, project managers who have spent the greater part of their lives doing them have their own eclectic ideas of what constitutes a successful software project. The three most pertinent here are that in the era of slide rules, producing ten lines of Fortran (now Ada) code a day was about the right speed; that a workday was eight hours long; and that the average project was expected to take about four years to complete. [...]

These Lockheed stalwarts of the military suddenly found themselves managing projects that not only had to work right the *first* time, but also had to be *fun* and had to be completed in about *six months*. For the coders [...], a typical workday might have been 20 hours. Traditionally most of those hours were at night. Perhaps only a few hours coincided with the project manager's workday. This fact alone made traditional supervisory techniques, rooted in a nineteenth-century prospectus of continuous visual surveillance, maddeningly impractical. The coders wrote in assembly language and, in the case of the occasional rabid *aficio*-

nado, in machine language. They employed every arcane trick imaginable to circumvent the obscure problems that plagued all the first- and second-generation microprocessors and thereby to make their code run faster and snazzier – tricks like self-modifying code, and code that took advantage of undocumented traps and weirdnesses in the chips' subcode”.

Stone, 1995, pp.129-131

Until the 1990's it was believed that the basic process of developing high-quality software was more or less the same for teams of computer professionals as for hackers, despite differences between commercial and hacker software development goals. Eric Raymond has been described as “the Internet hacker culture's tribal historian and resident anthropologist”. He stated that he 1993, after 15 years of writing software, had “absorbed all the conventional engineering wisdom we know from Fred Brooks and other writers in the software engineering tradition: Keep your project groups small, keep your objectives well-defined, keep everything tightly controlled, obsess about bugs and so on” (Dern, 1998). That was then.

Open source software development

Open source software development (hereafter also called “open source”) turned everything that was known about software development upside down. One definition of open source states that “open source is basically software developed by uncoordinated but collaborating programmers, using freely distributed source code and the communication facilities of the Net” (Dyson, 1998). A more technical definition, “The Open Source definition⁵”, consists of nine different criteria and specifies rules for free redistribution, the status of derived works, the integrity of the author's (sic!) code etc.

The development of the open source model and the very term “open source” is recent and it is not until the very last years that anyone has written about open source and its principles. The model itself rose to fame together with the Linux operating system that began to be developed in the beginning of the 1990's. The development of Linux was started/run/guided by the (at the time) Finnish university student Linus Thorvalds. Open source however is larger than Linux and in summing up the open-source movement, O'Reilley (1999) states that “The Internet [...] has grown out of the open-source software community”. An example of this is the Apache web server software. Not only is it available for free, but it is in fact the most popular software for web servers on the Internet with a market share of over 50%. And, “in July [1998], IBM announced that it was joining the Apache Group, [...] dedicating a team of programmers to helping to develop Apache on the [Microsoft Windows] NT platform” (O'Reilley, 1998).

The best texts on the subject of open source are written by Eric Raymond and Tim O'Reilley. Eric Raymond has written about his experience of both having led his own open source project and having studied other projects, especially Linux. He developed his views on open source in an influential paper called *The cathedral and the bazaar* in 1997. This paper is credited with having influenced Netscape to make the source code for its flagship product Communicator freely available in early 1998. This paper and a number of his essays have been collected in a book published by O'Reilley and Associates, *The ca-*

5 The Open Source definition can be found at <<http://www.opensource.org/osd.html>>

thedral and the bazaar (Raymond 1999) but almost all of his papers are – in the spirit of the open source movement – also available on his homepage⁶.

Tim O'Reilley is the founder and CEO of O'Reilley and Associates, a book publisher who publishes (predominantly technical) books about open-source software and other computer-related topics. He has been promoting open-source software and he organized the first "Open Source Summit" in 1998 where the leaders of the open-source community met. (One of the results of the summit was that the term "open source" was adopted as a generic term, thereby replacing the unclear and ideologically loaded earlier term "free software".)

The Open Source Initiative (<<http://www.opensource.org>>) is another source of information. The Initiative has the explicit agenda of convincing as many as possible of the excellence of the open source model.

What then is the difference between open source and traditional software development? Raymond describes the difference as a clash between two different cultures, where the traditional approach of developing software resembles that of building a cathedral while open source constitutes a bazaar of different agendas that still, for some reason, functions:

"I also believed there was a certain critical complexity above which a more centralized, *a priori* approach was required. I believed that the most important software [...] needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages⁷ working in splendid isolation [while] the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches".

Raymond, 1997/1999, p.29-30

Open source is not something brand new in itself however. Some of the principles of this development model have been part of the way software has been developed in the hacker community for a long time. "Some of the most significant advances in computing, advances that are significantly shaping our economy and our future, are the product of a little understood "hacker culture" " (O'Reilley, 1999). What is new with open source is the name, the concept that it is possible to develop even the most complex types of software (i.e. even operating systems) with this type of organization and the development of an economic model that explains and justifies open source also in economic terms. With that comes a second wave of theories regarding for what purposes and in which situations open source is applicable, what open source means to business models, for jobs in the computer industry, for software quality and on the way software is developed.

Open source versus SvenskMud

This chapter has so far described software development over a period of 50 years in a handful of pages, to finally lead to how software is developed in SvenskMud. The fact is that before I encountered the scarce but growing literature on open source software, nothing else about software development even started to describe a process that resembled the one I had observed in SvenskMud⁸. If anything at all deserves to be described as "a

6 <<http://www.tuxedo.org/~esr>>

7 Do take note of the fantasy-inspired terms "wizards" and "mages".

8 I have observed SvenskMud since 1996, but it was not until 1998 that the open source concept came to public attention and in 1999 when it came to my personal attention through written material. *The cathedral*

great babbling bazaar of differing agendas and approaches”, it is SvenskMud! Even at a cursory glance, it is clear that there are many similarities between open source and the process of software development in SvenskMud.

I will not get into a lengthy comparison between the genealogy of the principles that are common to open source, to muds in general and SvenskMud in particular. It is sufficient to point out that SvenskMud was started in July 1991 and is by now one of the oldest continuously running muds on the Internet⁹. As has been mentioned, software development in SvenskMud and open source both fall back on or incorporate principles from a hacker culture that are much older than either one.

*The cathedral and the bazaar*¹⁰ is written as a tale that unfolds over time about the author “trying to understand why the Linux world not only didn’t fly apart in confusion but seemed to go from strength to strength at a speed barely imaginable to cathedral-builders” (ibid., p.30). This effort took more or less three and half years. Another parallel strand in the paper is the tale of the author’s subsequent attempts at managing a software project, “Linux-style”. These two strands are interspersed by a third narrative element that is a series of lessons learned by the author as the course of events unfolds.

The 18 lessons formulated by Raymond in the paper are at the same time proposed as theories of software engineering. Some of the lessons are older than Linux, but are in the paper all presented in relation to the developments of Linux¹¹. The enumerated lessons/theories of the paper are amenable to present and compare – theory by theory – with how SvenskMud is being developed. I will look at the first eight principles and compare them head-on. Where nothing else is specified all the numbered principles below as well as all quotes come from *The cathedral and the bazaar* (Raymond, 1999).

Of the other lessons – lessons 9 to 18 – some are too general, some too technical and yet others too geared towards developing a type of software that solves a specific problem for its users, which SvenskMud is not. Lesson number 17 for example is very general; “A security system is only as secure as its secret. Beware of pseudo-secrets”. Lesson number 9 is a lesson about programming, not about (social aspects of) systems development; “Smart data structures and dumb code works a lot better than the other way around”. Lesson number 13, “perfection in design is achieved not when there is nothing more to add, but rather when there is nothing more to take away” implicitly assumes that the whole endeavor of developing a program is instrumental and has a specific goal. This is true for most programs, but not for open-ended games, like SvenskMud.

and the bazaar was written in 1997, but most other papers on open source were published in the second half of 1998 or later.

9 Another mud was started two years earlier, in May 1989, at Lysator (the academic computer club at Linköping University). This mud became NannyMud (<<http://www.lysator.liu.se/nanny/>>), which supposedly is the oldest mud running on the original (but modified) LP-mud code. In the early years of SvenskMud’s history there was a cross-fertilization from NannyMud to SvenskMud both in terms of ideas and in the people involved.

10 The book “The Cathedral and the Bazaar” was published in 1999. I here refer to the original paper that was written in 1997.

11 It would be wrong to call the lessons in the paper principles of open source software development, as the term “open source” had not been coined at the time the paper was written. But, *The cathedral and the bazaar* was in fact seminal in bringing the underlying principles to attention and in forming the concept of open source software.

Raymond's lessons

1) *Every good work of software starts by scratching a developer's itch.*

“Too often software developers spend their days grinding away for pay at programs they neither need nor love. But not in the Linux world – which may explain why the average quality of software originated in the Linux community is so high.”

This is a very important principle both in open source and in SvenskMud. O'Reilley (1999) further explains that:

“One of the problems of commercial product development is that the focus is on creating products that will make a profit for their creators. There are many products that meet real needs that are either too specialized or too early to market to produce the return on investment that [business] demands. *In fact, many open-source projects have been started to solve a user's particular problem*” (my emphasis).

An example of this is Sendmail, a program that is part of the underlying infrastructure of the Internet and that forwards electronic mail across heterogeneous computer networks. The precursor of Sendmail was written by Eric Allman to solve Eric Allman's specific problem at UC Berkely 20 years ago. Another example is the World Wide Web, developed by high-energy physicists Tim Berners-Lee so that he and other high-energy physicists could share their work with each other. A third example is Linux. Linus Thorvalds wrote about his original goal for developing Linux, stating that; “at first I just wanted something that would run on my 386” (Thorvalds, 1999), i.e. his PC with an Intel 386 microprocessor.

Every part of SvenskMud's software is written because something scratched a developer's itch. This is true for SvenskMud on no less than three different levels. It is true for the LP-mud code that SvenskMud is based on, for the original developments of SvenskMud itself and for everything developed inside SvenskMud since. Some pieces of software are never finished or incorporated into SvenskMud exactly because the original itch does not itch the individual developer enough any longer. Lars Pensjö states several reasons for his interest in creating the initial LP-mud and what is noteworthy is that *all* reasons are technical and none is social:

“What drove me to do LP-MUD was to start with above all a technical interest. The program consists of quite a few components, for example compiler technology, advanced data structures, realtime functions, object-oriented solutions, interprocess-communication etc.”
[...]

The idea that the game could be extended by other [developers] did not exist to start with. It arose before long when I realized that my own fantasy would not suffice to create a comprehensive game.”

Excerpt 6.1, Pensjö 2000, personal communication

The development of SvenskMud was started/run/guided by a student at Linköping University, Linus Tolke. He felt there was a need for a Swedish-language alternative to all the English-speaking muds on the Internet. That was his itch. Every other SvenskMud magician-programmer had to start his or her careers in SvenskMud as players. There are no exceptions and no shortcuts to this rule. It does not matter if a new player is a master pro-

grammer in another mud and he or she could make a substantial contribution in terms of quality computer code immediately. In order to become a magician in SvenskMud, each person has to make an investment in time and effort and play SvenskMud extensively. And, playing SvenskMud extensively is a way of ensuring that to-be magicians are provided with much SvenskMud-specific knowledge and many SvenskMud-specific itches.

Just as with Linux and other open source software, all software development in SvenskMud is done on idealistic (non-profit) principles. All time spent in SvenskMud (both as a player and as a magician) is spent because it is enjoyable and preferable to other activities.

Despite Raymond's statement regarding the high quality of the Linux code, enthusiasm alone does not automatically ensure that the written code is of good quality. Some magicians who write code in SvenskMud are novices or inexperienced programmers when they become magicians in SvenskMud. Others have studied programming, perhaps at high school or by themselves, but have had little practice before they encounter SvenskMud. It is probably true in the case of Linux. The Linux archive site accepts contributions from anyone, but surely not anyone's code is incorporated into Linux.

2) *Good programmers know what to write. Great ones know what to rewrite (and reuse).*

“An important trait of the great [programmers] is constructive laziness. They know that you get an A not for effort but for results, and that it's almost always easier to start from a good partial solution than from nothing at all.”

Raymond adds that Linux used “code and ideas from Minix, a tiny Unix-like OS for 386 machines. Eventually all the Minix code went away or was completely rewritten – but while it was there, it provided scaffolding for the infant that would eventually become Linux.”

This “constructive laziness” is built into not only SvenskMud, but also every single mud on several different levels. When Linus Tolke started SvenskMud in 1991, he did not write all the code for the mud himself. What he did was to download the source code for running a mud from the Internet. For an LP-mud, that would consist of a “mud driver” and a “mudlib”. The mud driver is the mud server program, written in UNIX and responsible for basic functions like communication between the system and the mud users. The mudlib is a database with “building blocks” – written in a mud-specific variant of the C programming language called LPC – which SvenskMud magician-programmers use when they program.

What Linus Tolke then proceeded to do was to adapt the mud driver to the Swedish language and to the computer it was going to run on. Finally he translate some of the “public” areas in the mud, i.e. the descriptions the players could see of these places, from English into Swedish. As part of Linus' own “constructive laziness”, he then proceeded by mustering others to help develop the content of SvenskMud and for his own part concentrated on developing and altering the mud driver when needed, i.e. working on the most basic technical level of the mud.

For muds in general, every single mud system which has been developed during the last 20 years is built, if not directly on code, then at least from ideas of previously existing mud systems. LP-muds are only one of many types of mud, and each member can exist in a variety of versions. Anyone who wants to start a new mud only needs to choose the mud

system he (or more seldom – she) prefers, and download the source code which is publicly available at no cost. As can be seen, the “constructive laziness” is active on a higher level than SvenskMud too. Shah and Romine even describe how large chunks of mud content, for example in the form of stock castles (i.e. whole areas created by a magician/programmer) have been transferred between muds; “This is why you see Morgar’s Castle and Kantele’s Mansion on several different Muds.” (Shah & Romine, 1995, p.209.)

Another level where the “constructive laziness” is at work in SvenskMud is on the level of the individual magician-programmer. All code of all magicians is open to inspection by any other magician in SvenskMud. Magicians are encouraged to look at other magicians’ code, especially if they are beginners. One of the then spoof commandments in the handbook for SvenskMud magicians (Tolke, 1993) states that “You shall steal”¹². What does this mean? It means that “if you don’t know how to do certain things, then try to recall where you have seen something similar and check out how it is done there. The best way to learn is to see how others [more experienced magicians] do things”. (Tolke, 1993, p.12). The first recommendation for magicians of the handful that exist in SvenskMud gives advice to (novice) magicians on how to go about to build things in SvenskMud:

“Cooperate!

Always use objects that already exist to do different things. A suitable way to do this is with the instruction “inherit”. LPC is an object-oriented programming language with multiple inheritance.

- If there are no ready objects [that do what you want to do] then talk to a magician who has already created such an object so that both of you will use the same object (it saves time and energy!).”

Excerpt 6.2, RULES, 1997

3) *Plan to throw one away; you will, anyhow*

Quoted from the title of chapter 11 in Frederick Brooks’ book on software engineering (Brooks, 1975/1995), lesson number three means that “you often don’t really understand the problem until after the first time you implement a solution. [...] So if you want to get it right, be ready to start over *at least* once”. Raymond notes that “in a software culture that encourages code-sharing, this is a natural way for a project to evolve”.

This lesson is a little difficult to apply in the context of SvenskMud *and* open source. Brooks meant that any first solution would always be unsatisfactory and the choice at that point was either to start over and build it *right* or try to salvage and deliver the unsatisfactory version to the customer. However Brooks assumed that there was an ultimate point when software developments have to stop so that software delivery could be started. However, in the case of both Linux and SvenskMud, the computer program is rather in a perpetual state of change and parts of the program are being developed or replaced continuously. To constantly “throw one away” is built into the very model of an incremental and continuously developing open-source project. Brooks himself recognizes and admits, 20 years after the original lesson was formulated, that “the biggest mistake in the “Build one to throw away” concept is that it implicitly assumes the classical sequential or waterfall model for software construction”. (Brooks, 1975/1995, p.265)

12 The original commandment reads: “You shall not steal” (Jerusalem Bible, 1966).

On the other hand, hacking away in SvenskMud is never a finished process as SvenskMud is always up and running and never finished. A magician can choose to work with whatever project he or she likes, including developing someone else's (abandoned) code. This could perhaps be construed as starting over with that code/project/program.

Since SvenskMud is a babbling bazaar of differing agendas and approaches, it is very much up to the individual magician if he sees his first attempts as a finished product or as a barely usable first attempt:

“What I've coded? [I] Started on an area that I have to say is a shame that it is out in the mud since it contains a little bit too many bugs in my opinion. Fixing it is my intention (has been for 2 years now), the problem when I code is that I myself think [that] what I code is so “funny” that I quickly want to get it out [into the mud], then perhaps you suffer from the Microsoft-syndrome, that you let the users/players bug test it all.”

Excerpt 6.3, WQ #2.2

On the other hand, two different magicians have told me during interviews that they chose to reprogram two major projects in SvenskMud from the beginning after their initial, feeble attempts (the sorcerers' guild and the winter wolfs' guild).

An important principle that differs SvenskMud from other software projects is that old code, sub-standard code, any code once implemented in SvenskMud is respected to a very high degree. Old code is basically never thrown away although it might be moved to a less central part of the SvenskMud world. A guiding principle is that old code can be improved if needed, but never removed. Old code is thus complemented or “upgraded”, but never thrown away. Gerald Weinberg (1971/1988, p.53) has commented that “[the programs of programmers] often become extensions of themselves” and continues:

“Well, what is wrong with “owning” programs? Artists “own” paintings; authors “own” books; architects “own” buildings. Don't these attributions lead to admiration and emulation of good workers by lesser ones? Isn't it useful to have an author's name on a book so we have a better idea of what to expect when we read it? And wouldn't the same apply to programs? Perhaps it would – if people read programs, but we know that they do not. Thus, the admiration of individual programmers cannot lead to an emulation of their work, but only to an affectation of their mannerisms.”

Weinberg, 1971/1988, p.53

As to the SvenskMud code, it is probably safe to say that no-one reads the programming code like a person reads a book. However, SvenskMud players *do* read (or rather use, or experience) the code again and again when they *play* “SvenskMud – the game”. And so we can see an interesting connection between mud code and the literary works of an author. We generally think it is abominable to burn books – even bad books. In SvenskMud, the principle is to not throw away any code – even bad code (as programming code goes). Once completed, each individual magician's code enjoys a high (but not absolute) level of integrity and it resides in his or her own “catalogue”. And there it remains to his or her eternal fame or shame.

In the handbook, the very first paragraph of text purports to describe not the goals of SvenskMud, but its “place in the world of literature”! In the very first section, Tolke (1993) uses the metaphor of the mud being a “book” which is being “read” by many “readers” at the same time as he explains what is special about muds and SvenskMud.

4) *If you have the right attitude, interesting problems will find you.*

There is actually no explication of this lesson in Raymond's article. Inasmuch as "the right attitude" is important, Raymond's whole paper outlines what that attitude is, or should be.

5) *When you lose interest in a program, your last duty to it is to hand it off to a competent successor.*

This lesson obviously has nothing much to do with Linux on the most general level as it has always and still is managed by Linus Thorvalds, but all the more to do with other open source projects including Raymond's own experiment to manage a software project "Linux-style". He "inherited" the responsibility of a program when the person who was previously responsible thought it was in the best interest of the program. This lesson makes sense, but it does not work very well in SvenskMud. Many magicians have the very best of intentions; to complete their grand project(s), to start new projects, to continue to support their code, but they slowly drift away from SvenskMud, never to look back on the code they left behind. To understand why, some comparisons can be made between how bugs are reported in Linux and SvenskMud.

In both Linux and SvenskMud, the individual contribution of a programmer becomes part of a greater whole that the users experience. However in SvenskMud, the principle of the user illusion (see chapter 4) becomes an obstacle; the player cannot pinpoint the exact magician-programmer who is responsible for the code that created the specific problem he or she encountered while playing SvenskMud. In fact, the player is not supposed to think of SvenskMud as a computer program at all, but rather as a world that he or she is inside (although that illusion quickly breaks down when the player encounters a bug).

In Linux, on the other hand, the name and e-mail address of a developer remains, and that person will continue to get bug reports by e-mail until a problem is fixed or until someone else takes over the responsibility. By contrast in SvenskMud, the individual magician first has to create special logging functions to find out if the players use objects he or she has created in the first place, and that magician actively has to log in to SvenskMud to see if and what bugs have been reported.

The conclusion is that it is easier to fix a bug in Linux than not to (since you will continue to get bugged about it otherwise). In SvenskMud, on the other hand, it is more of an effort to fix a bug than not to.

6) *Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.*

"Users are a wonderful thing to have, and not just because they demonstrate that you're serving a need, that you've done something right. Properly cultivated, they can become co-developers. [...] Given a bit of encouragement, your users will diagnose problems, suggest fixes, and help improve the code far more quickly than you could unaided."

This lesson is easiest to live by when it comes to software developed *by* hackers *for* hackers. In the extreme cases, the whole user base consists of potential developers who not only suggest fixes but also provide replacement code when they encounter errors. This was the case for both Linux and SvenskMud at their respective inception. When SvenskMud

started in 1991, the user base (of both players and magicians) mainly consisted of university students at technical faculties, with an interest in computers and programming. Since then, new groups have access to the Internet and many Linux and SvenskMud users would today not be able to “suggest fixes” or “help improve the code” even if their life depended on it.

Although not all SvenskMud players know how to program, all can report problems they encounter. And some do have the technical knowledge to suggest ways to fix them. However, there are two issues worth mentioning that differ SvenskMud from Linux.

The first issue is the ideal of maintaining a user illusion in SvenskMud. Not only do some players not have sufficient knowledge to suggest fixes, but unlike Linux, the ideal in SvenskMud is that things remain exactly that way. The players are not supposed to know about or understand what is behind the “curtains”; the technical system they experience when they play SvenskMud. One of the spoof commandments from the SvenskMud handbook reads “you shall bear false witness against your neighbor”¹³:

“The whole world [i.e. SvenskMud] is of course really an illusion. See to it that you keep up the appearances on behalf of the players as long as possible. Avoid sending text of technical nature to the players. [...]

Be especially careful if you echo or shout things [i.e. transmit messages to everyone who is logged in to the system]. Take into consideration that the players live in another world and adapt your messages in accordance. Shout for example “now I leave for a long expedition in east and north” instead of “now I’ll go home to bed”.”

Excerpt 6.4, Tolke, 1993, p.13

The second issue is that players and magician do not necessarily have the same goals in SvenskMud – unlike Linux users and developers¹⁴. It has happened several times that when players find an error they can use for their own purposes, they do that instead of reporting the “bug” (see chapter 9). At other times though, players can do substantial contributions that are of great help to the magicians in debugging and developing svenskMud. One player who helped the magicians to diagnose bugs got a specially “minted” medal for his services, which he could proudly display to other players in the game. It should be noticed that unique objects are unusual in SvenskMud, comparable perhaps with the status that handcrafted or custom-made objects lend in this day and age of machine-produced similarity.

Naturally the majority of bugs are not such that the players can *use* them. Most problems just make SvenskMud function less good than it could and in these cases the goals of players and magicians are the same; to get SvenskMud to function better.

7) *Release early. Release often. And listen to your customers.*

“Early and frequent releases are a critical part of the Linux development model. Most developers (including me) used to believe this was bad policy for larger than trivial projects, because early versions are almost by definition buggy versions and you don’t want to wear out

¹³ The original commandment reads: “You shall not bear false witness against your neighbor” (Jerusalem Bible, 1966).

¹⁴ Different players have different motivations and goals though, and *some* players do share with the magicians the ideas of what makes SvenskMud a good mud.

the patience of your users. This belief reinforced the general commitment to a cathedral-building style of development. If the overriding objective was for users to see as few bugs as possible, why then you'd only release one every six months (or less often) and work like a dog on debugging between releases.”

“When you start community-building, what you need to be able to present is a plausible promise. Your program doesn't have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.”

This is a lesson from the development of Linux that breaks with all previous known principles of developing software. Raymond continues: “In [1993] it wasn't unknown for [Linus Thorvalds] to release a new kernel more than once a *day!* [...] Linus was keeping his hackers/users constantly stimulated and rewarded – stimulated by the prospect of having an ego-satisfying piece of the action, rewarded by the sight of constant (even *daily*) improvement in their work. Linus was directly aiming to maximize the number of person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout”. To release a new version of an operating system once per day is a tempo that is unheard of before Linux. The tempo is considerably less hectic today, but still very fast compared to the development of comparable products (i.e. operating systems like Microsoft Windows 2000, Windows NT, Sun Solaris etc.).

SvenskMud again differs too much for a straight-off comparison to make sense. SvenskMud is a shared environment that is up and running 24 hours per day while new versions of Linux is downloaded from the Internet to run on individual desktop computers or servers. SvenskMud – just as all LP-muds – consist of different levels of code. The SvenskMud driver and mudlib change at times, but far from as often as every day. Changing the driver or the mudlib is akin to changing the physical laws of the svenskMud world.

An individual magician can on the other hand change or append new code (new content in the SvenskMud world) or fix bugs any and every day. If the magician has the authority, he or she can load a new version of the code that replaces the old version on the fly, and any changes are immediately activated in the game. This means that players can use new SvenskMud code immediately as it is finished and that the turn-around time for magicians to see new code incorporated into SvenskMud is gratifyingly short. In this respect, as to the contents of the game, SvenskMud could change every day.

However, the scale and the tempo of development is *very* different in SvenskMud compared to Linux with its millions of users and thousands of developers (Thorvalds, 1999). SvenskMud is a small local project in comparison with Linux, which uses the entire Internet as the potential user/developer base. It is my understanding that there has been, at an average, half a dozen magicians or so who actively write new code for SvenskMud (as apart from fixing bugs and maintaining the already existing code) at any specific point in time during the last few years¹⁵.

This lesson, to “Release early. Release often. And listen to your customers” might or might not be applicable to SvenskMud depending on how the lesson is interpreted. How-

¹⁵ Near SvenskMud's inception, at the time the handbook for magicians was written, the number of magicians who produced code were as many as 20 (Tolke, 1993, p.2).

ever, Raymond states that the *reason* Linux was released so often was because “Linus was directly aiming to maximize the number of person-hours thrown at debugging and development”. I would say that this is valid in SvenskMud too, albeit on a different scale and using different means. The fact that no one in SvenskMud has extraordinary powers to decide over sensitive matters – to act as a police of good taste and of worthy endeavors – and that such issues are always treated with a very light touch can be seen as an indirect attempt to maximize the number of person-hours thrown at debugging and development in SvenskMud. A more forceful policy would perhaps attain a more stringent SvenskMud world or a more stringent code base, but at the expense of attracting fewer magician-programmers. The issues are an instantiation of a long-standing conflict between, on one hand, freedom and, on the other hand, control.

8) *Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.*

”Or, less formally, ‘Given enough eyeballs, all bugs are shallow’”. The idea is that open source code means that programs are used and tested in many contexts, leading to bugs being fixed quicker and in the end, to more robust software. Debugging is parallelized and the person who notices a problem does not necessarily have to be the person who fixes it.

This general observation has not been unknown before open source and Fredrick Brooks mentions that the cost of maintaining a program is dependent on the number of users of that program. More users mean higher costs of maintenance because more users find more bugs (1975/1995, p.121). However before Linux, this was regarded as a potential downside of a successful program, a cost in terms of time spent maintaining it. With Linux, this relation was elegantly transformed from a *problem* (of maintenance) into an *asset* (in the development process). Raymond emphasizes that this signifies a huge difference in the way software bugs – and by extension the way computer programs – are regarded by the two different approaches:

“Here, I think, is the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. [...] In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena – or at least, that they turn shallow pretty quick when exposed to a thousand eager co-developers pounding on every single new release.”

The contrary view, a view that is representative of the cathedral-builder mind set and that also relates to the previous lesson (lesson 7), is expressed in the very first paragraph of an article entitled “How good is good enough? An ethical analysis of software construction and use”:

“Perfect software for a complex system cannot be guaranteed in practice. Every significant piece of new software can be assumed to contain errors, even after thousands or millions of executions. Software that matches its specifications perfectly may contain errors since specifications are not necessarily error-free. Software that matches its specifications perfectly and has perfect specifications may be used erroneously by users. Thus the question of how good is good enough relates to when to release the software, and also how to safeguard the actors from the inevitable errors.”

Collins et al., 1994, p.82

But had bugs been a “deep” problem in need of a specialized cathedral-like tight team of developers to solve them, then Linux “should at some point have collapsed under the weight of unforeseen bad interactions and undiscovered “deep” bugs” (Raymond, 1999, p.42).

Had bugs indeed been “tricky, insidious, deep phenomena”, then SvenskMud would also have collapsed a long time ago, under the weight of its many programmers and its 3 million lines of computer code. One powerful way to master the complexity of having many parallel programmers and internal chains of dependencies is to modularize the software. SvenskMud (and Linux) consist of many modules which are relatively independent of each other and which interact with each other in ways that are predetermined and tightly defined. The open source model of software development with parallel development and loose control more or less requires this mode of operation to be effective. McKelvey writes on the characteristics of this model:

“The software becomes malleable and easy to change, as it is open to user/developers with the competence to program. Some disadvantages are that the quality may vary from excellent to terrible, with costs moved to the user/developer to test and trial the programs. [...] In addition, the software may not be tightly integrated into one bundle, so that while flexibility enables adaptability, it may also lead to confusion and difficulties of getting everything to work together.”

McKelvey, 2000

Replacing developer with magician and user with player, all of the above is true also for SvenskMud. The drawback, as noted, is the potential disunity of “SvenskMud – the computer program” as a whole. A practical example will highlight this.

Let us say that a magician-programmer develops a fishing rod that works in the lake just beside the place where a player finds it (perhaps the lake is also created by the same magician). Now, let us say that the player takes the rod, walks to another place in the mud and finds another lake. In the worst case, the fishing rod object (i.e. the small computer program that determines the behavior of the fishing rod) would not work in the new lake because the programmer did not think of this possibility and did not make the object general enough to work in situations other than the intended. In an even worse case, the lakes themselves would not even work in the same ways, i.e. different magician-programmers had had different ideas about the purpose of water (in the context of the game) and had implemented different rules for how water behaves/works in the SvenskMud world.

A much more tricky case would be if a user tried to use the fishing rod for some other purpose than fishing, for example using the rod as a rapier or trying to break the rod in pieces to use it to make a camp fire. It is easy to think of many other complex and tricky problems that have to do with one programmer’s code interacting with another programmer’s code in SvenskMud¹⁶. An especially taxing function for SvenskMud programmers is the use of “shadows” [skuggor]. A shadow monitors communication between the player and the mud program and at times alters that communication in game-related terms. Let us say that a player can wear a magic amulet that makes him more perceptive of hidden

16 Some of the most heralded anecdotes in SvenskMud deal with occasions when the code of different objects (created by different magician-programmers) have clashed, resulting in bizarre and totally unexpected behavior of the game environment. An example is the stone guild members’ phobia for different animals (one phobia per player), and the stone guild member who happened to chance upon an airship whose name contained the animal he could not bear; a goat [“M/S Geten”].

dangers in a room. Let us further say that a player can don a helmet that limits his field of vision affected so that he might not notice all the things another player in the same room would notice. Both of these functions would typically make use of a shadow to act as an extra layer of the interface between the player and the mud. Making use of a shadow can become especially taxing when two shadows interact with each other such as when a player both wears a magical amulet and a helmet. The interaction between different sequences of code can technically become very complex and difficult to keep track of. Which shadow should take precedence? Should it depend on the order in which the player “dons” the objects/shadows or by some other principle?

From such examples it is also easy to see the connection between, on the one hand, different ways to structure the development process and different ways to structure “SvenskMud – the computer program” and, on the other hand, programming SvenskMud as world creation, as imposing (a particular, specific) order in the SvenskMud world.

Just as in some previous lessons, the players do not have the knowledge and are forced to live in ignorance of the relationship between a bug/problem they encounter and the programming code. Therefore their attempts to help might not be of so much help to a magician who tries to diagnose the problem. The community of SvenskMud magicians can fully work together as a co-developer base, according to the criteria Raymond develops; one magician (or player) can find a problem, another magician can diagnose the problem correctly and a third can implement a solution. “The dog that could retrieve castles” in chapter 4 is an example of this.

Differences between open source and SvenskMud

There are some characteristics that make software development in SvenskMud different from open source. In some cases the characteristics make such a difference that some of Raymond’s lessons are not applicable and have to be modified.

One characteristic is that SvenskMud is not instrumental in the way ordinary programs are. Ordinary open source projects, be it an operating system, a web server, a programming language or an application, all have in common that they are tools, developed in order to achieve a specific function. Linux, as an operating system, acts as an intermediary between the computer hardware and application programs. Linux developments are also driven by actors and actions outside the Linux development community, like end-user needs and technical developments in the computer industry. In the end, there are just some tasks that absolutely have to be met by Linux and the Linux developers in order to fulfill the requirements of a full-blown operating system.

Not so with SvenskMud. SvenskMud does not have a fixed purpose, or at least not a purpose that can be operationalized. In one of the first pages in the handbook for magicians, the goal of SvenskMud is presented as “to offer an alternative to the English-speaking American-influenced culture that exists among hackers and especially among muds” (Tolke, 1993, p.2). The diversity of the different agendas and approaches is wider in SvenskMud because of the type of software it represents *and* because of the self-contained (or isolated) status of the SvenskMud project in relations to the larger world (of computers and computing) outside SvenskMud. Unlike some other on-line games, SvenskMud is not even developed in order to earn money in any way whatsoever.

Since the handbook was written 1993, an informal hierarchy of purposes or goals of SvenskMud has been defined and presented at several SvenskMud meetings I have at-

tended. While there are different goals for players and magicians, my conclusion is still that the very most basic of SvenskMud's functions is to entertain; to entertain players who like to play SvenskMud and magicians who like to program. The lack of a more specific purpose makes programming in SvenskMud look more like an activity, a hobby, a pastime, than a goal-oriented task (c.f. Linux). Programming in SvenskMud is programming for its own sake and it is furthermore *very* far from industrial software development (despite, or perhaps *because* most magicians study or work professionally with computers). It gives a lot of freedom to the individual magician and that freedom in its turn lends an aura of bazaar-like chaos to the whole endeavor.

It is instructive to compare this with Raymond's lesson number 13 that was mentioned earlier, but never explicitly discussed. The lesson was that "perfection in design is achieved not when there is nothing more to add, but rather when there is nothing more to take away". Raymond further states that "When your code is getting both better and simpler, that is when you *know* it's right". The instrumental character of the program is implicit and the lesson should in fact read "perfection in design is achieved not when there is nothing more to add, but rather when there is nothing more to take away *in order for program A to perform functions X, Y and Z*".

In SvenskMud as noted before, code is in principle never taken away, only added. What is more important though is that there is no *goal* against which to compare the code in SvenskMud in order to decide if the code finally is right or not (e.g. "that is when you *know* it's right"). Magicians in SvenskMud spend more time creating new code than maintaining old code. In industry, the state of matters is the other way around, programmers spend most of their time not developing programs, but testing programs, debugging programs and correcting programming errors¹⁷. The SvenskMud way of doing things is understandable from the perspective of the individual magician-programmer, because it is more fun to realize your ideas in code than to look at old code of other magicians and correct it. This practice makes for a good hobby but a "bad" computer program.

Compare this with Brooks' (1975/1995, p.122) observation about maintenance of industrial software: "The fundamental problem with program maintenance is that fixing a defect has a substantial (20-50 percent) chance of introducing another. So the whole process is two steps forward and one step back". Furthermore "All repairs tend to destroy the structure, to increase the entropy and disorder of the system. Less and less effort is spent on fixing original design flaws; more and more is spent on fixing flaws introduced by earlier fixes. As time passes, the system becomes less and less well-ordered. Sooner or later the fixing ceases to gain any ground" (ibid., pp.122-123).

After almost ten years of continuous development and with the number of developers reaching over 100, SvenskMud is more like a patchwork quilt than a consistent program, and it is a wonder that SvenskMud does not crash under its own weight. Especially taking into account that "structure" has never been a prioritized issue in SvenskMud. Despite this, every now and then a (new) enthusiastic young magician suggests to make SvenskMud more realistic, be it by introducing day and night, temperature, seasons or weather, a more realistic battle system or something else. What these magicians do not realize is that in order to add realism with a change like this, the change would need to be implemented retroactively throughout *all* the SvenskMud code. Throughout all *old* code. Throughout

¹⁷ According to Raymond (1999, p.143), no less than 75%, and perhaps as many as 95%, of all programmers work with maintaining old code rather than developing new.

ten years of old code. Furthermore, it would have to be decided which factors should be more important and which should be less important, and what factors should affect what other factors and in what way within the game¹⁸.

Not only does SvenskMud lack a specific purpose, it also lacks a proper project leader. Many of the functions of a project leader (i.e. to decide important matters) are absent or delegated in SvenskMud. Although the creator of SvenskMud still keeps a watchful eye on SvenskMud and has the formal authority to make the decisions in any and all matters, he most often has no strong opinions and has never played an active part in the operative activities. It is the individual magicians who decide what they want to create in SvenskMud.

Raymond remarks that “while coding remains an essentially solitary activity, the really great hacks come from harnessing the attention and brainpower of entire communities”. While both successful open source programs and SvenskMud have a group of programmers surrounding them, SvenskMud is more than “just” a group of programmers who have coalesced for a hack¹⁹. Before becoming a magician, a person first has to be properly socialized and go through the “initiation rites” of playing SvenskMud from beginning to end. And having become a magician, that person will then have the *permission* to program, but not necessarily the *ability*. Having become a magician, a person is allowed to program. It does not much matter if a new magician is inexperienced as a programmer – there are experienced magicians around who can give a helping hand. This differs from open source where the “only” thing a contributor has to do to be part of the project – to be part of the “community” in Raymond’s words – is to contribute quality code.

A final difference is the problem the user illusion constitutes for developing a program fully according to the principles of open source. The user illusion purports to hide “SvenskMud – the computer program” from “SvenskMud – the game” and players (are supposed to) only experience SvenskMud – the game. This means that the players can not do as much as they would have been able to do for the effective development of SvenskMud compared to if the computer code had been made more visible to them. The user illusion hides the computer code and this goes directly against the principles of open source software. The problem works in both directions. Not only can the SvenskMud users (players) not communicate with the developers (magicians) in the most effective way, but the developers are usually not users of the “SvenskMud – the game” any longer²⁰. As an individual, a magician can as time passes lose touch with “SvenskMud – the game” as the players experience it. As a community, the magicians are dependent on the few magicians that have more active contacts with the players and on new magicians who recently were players themselves in order to understand who the players are, what goes on among them, what they think and talk about etc.

18 Let us say for example that day and night is to be introduced to the game in contrast to the eternal day that is the present norm. For the sake of simplicity we assume that this would not involve sunrise, dawn, dusk and sunset, but only consist of the places in SvenskMud having two different descriptions; one for the day and one for the night. This would entail going through thousands of rooms in the game to add an alternative description for nighttime. Furthermore this simplistic solution does not take into account if certain objects in a room should be more difficult – or impossible – to see or use during night than during day, what the game-related consequences of this should be and so on.

19 From the new hackers dictionary (third edition). Hack: “An incredibly good, and perhaps very time-consuming, piece of work that produces exactly what is needed.” (Raymond, 1996, p.231).

20 Although it happens that some magicians create a new player in order to continue to play “SvenskMud – the game” or for example to be able to test their own code from the perspective of the players. This practice is however gently dissuaded in SvenskMud, but has not been made into a big point.

Software development in muds

Morningstar & Farmer (1991, p.284) divide implementation challenges they faced in Lucasfilm's Habitat – an early (mid-1980s) multi-user graphical virtual environment – into two sorts. The first challenge had to do with creating a technological platform that worked and the second had to do with creating and managing the Habitat world. It is only the first challenge that falls under the domain of traditional software development. In this section I will describe some characteristics of software development which are specific to muds (i.e. independent from open source or any other model for software development). They fall in-between traditional software development (developing a program) and management of the SvenskMud world (Farmer and Morningstar's second challenge), because in a mud, management is to a large extent done *through* code.

The most important factor that sets software development in muds (social virtual environments) apart from other computer systems is the close coupling between a social system and a technical system. Amy Bruckman (1992) was the first person to comment on this fact. After having been in contact with James Aspnes who in 1988 created the first social mud, i.e. the first mud that was *not* an adventure game, she concluded:

“The change in the software encouraged different styles of interaction, and attracted a different type of person. The ethics of the community *emerged*. The design of the software was a strong factor in shaping what emerged. [...] In the case of Tinymud [the mud system that Aspnes created], the technology *is* a social system. It is therefore remarkable that the social changes Tinymud caused were not intended by its founder. Aspnes writes that “this approach attracted people who liked everybody being equal.” Somewhat accidental features of the artifact combined with a process of *self-selection* [created] a community with a strong, shared set of values.”

Bruckman, 1992

As apart from what Bruckman writes, it is perhaps not “remarkable” that the social changes that emerged in Tinymud were not intended by its founder. Even with an understanding of the close coupling between the technical and the social system – which Aspnes did not necessarily have – the *unpredictability* of emergent, “chaotic” phenomena is precisely one of the characteristics that make them and mark them as emergent²¹. That is a mere detail though. What is interesting is Bruckman's conclusion that a mud system in use is at the same time both a technical system *and* a social system. If one of them changes significantly, so will the other in an open-ended, unpredictable dialectical dance (see figure 6.3).

21 In emergent phenomena, collections of units can, through their *interaction*, give rise to properties that are more than the sum of their individual contribution. In “nonlinear systems [...] changes are amplified, breaking up existing structures and behavior and creating unexpected outcomes in the generation of new structure and behavior” (Elliott and Kiel, 1997, p.1). An example of an emergent phenomenon is the growth of a plant, which consists of rather simple components exhibiting simple behaviors, but whose combined behavior is so complex that it may not be reducible to a mathematical statement. For applications of chaos theory (Gleick 1987, Stewart 1989) to the social sciences, see Elliott and Kiel (1997), Ferguson (1997) and Epstein & Axtell (1996). For a more popular account of “the new biology of machines”, see Kelly (1994).

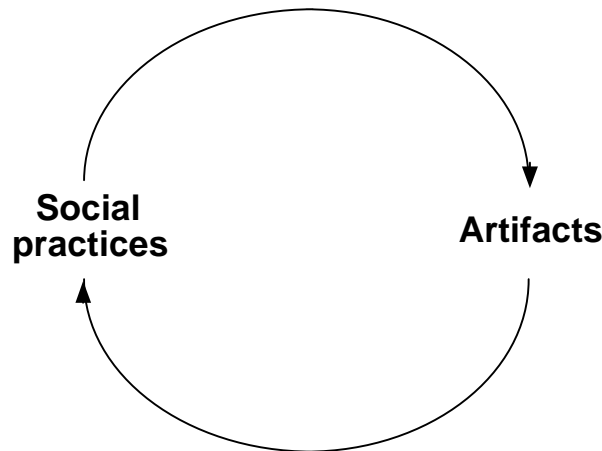


Figure 6.3. The social-technical design cycle. Adapted from O’Day et al. (1996). The relationship between the social system and the technical system in a mud is not causal, but dialectical and co-evolutionary.

Some examples of the close ties between a mud as a social system and a technical system are described in the introduction in O’Day et al. (1996):

“This paper describes the joint evolution of tools and social practices in Pueblo, a school-centered learning community supported by a MOO [...]. Examples illustrate how one can design and use a social practice to simplify a technical implementation, and how one can make a choice in technical implementation to work towards a desirable social goal. Social and technical practices in a network community co-evolve as social values and policies become clearer and as growth in the community pushed it toward changes in the distribution of authority and power.

O’Day et al., 1996, p.160

I illustrate this with an example from SvenskMud that also elucidates the unpredictability of the dialectics between the social and the technical system. It has been possible for players in SvenskMud to become engaged for a long time. Unlike many other mud systems, it has not been possible for players to marry each other within the mud though. In other words, there existed a *requirement* of being able to marry another player inside SvenskMud that stemmed from the social system, although I do not know how many players (if any) have specifically asked for this possibility throughout the years. As a response to this perceived – real or imaginary – requirement someone reprogrammed SvenskMud in 1999 so that players who were engaged could proceed to also get married. In other words, the technical system now offered a *possibility* that did not exist before. One player quickly seized upon this possibility in a way that greatly surprised the magicians and made them stumped. The player in question requested to be married to a motherly, matronly woman in the game, *Mother Karin*. But, Mother Karin is not another player but one of the SvenskMud non-player characters, i.e. a bot, or, a computer program. The magician who had created Mother Karin was left to make the decision on behalf of Mother Karin and

this later indeed did result in the marriage of a slightly reprogrammed (marriage-enabled) Mother Karin.

Some other examples of the close ties between the social and technical system have been described in “The lessons of Lucasfilm’s Habitat” (Morningstar and Farmer, 1991) and elsewhere. Some of the lessons Morningstar and Farmer preach are technical and others are social, but what they have in common is that they always relate to each other. To Morningstar and Farmer, the purpose of the technical dimension is to facilitate the social dimension, which in turn affects further technical developments. One of their lessons – in their own words the “possibly most controversial” assertion – is that “detailed central planning is impossible; don’t even try” (ibid., p.285). This lesson is quite in line with earlier observations of the emergent or evolutionary nature of these systems. Such characteristics are visible for example when O’Day et al. (1996) establish that “the system is always in flux, as the implications of design and use are absorbed and proceed to transform other parts of the system” (p.161).

Another lesson of Morningstar and Farmer’s (1991, p.294) is to “work within the system”. “Wherever possible, things that can be done within the framework of the experiential level should be. The result will be smoother operation and greater harmony among the user community. This admonition applies to both the technical and the sociological aspects of the system.” The experiential level Morningstar and Farmer refer to is basically the same as the one I refer to when I use the term “user illusion”.

What this all means is that muds – social virtual environments – constitute a special type of software that differs from other types. As was mentioned in the beginning of this chapter, there are different ideals that are desirable when software is developed, such as reliability, ease of maintenance, ease of modification, generality, ease of use and efficiency. These ideals are mutually unattainable in relation to demands in terms of time and costs, and all design choices are also choices regarding which of these ideals are more important than the others in a specific situation. On top of these always-present criteria of software development are some ideals that are specific to SvenskMud; a consistent theme of Sweden in the 19th century, references to Swedish culture and phenomena, a consistent use of the Swedish language throughout the mud, a programming/social environment that encourages magicians to develop new content, an entertaining environment for the players, a wall-to-wall user illusion and a high integrity of already-written code. These ideals, just as the general ideals for software, are often mutually unattainable and, at times, in direct conflict with each other.

What is special in muds is the fact that by redesigning the technical system, the social system within the mud is automatically changed. If some behavior appears among the SvenskMud players, that for some reason is judged to be undesirable by the magicians, then the technical system can be changed to counter this. What makes muds and other social virtual environments unique is the tight coupling between the technical and social system – an effect of the fact that a mud in use is a social system *within* an artifact. The artifact effectively determines both the possibilities and the constraints of the micro-society in question. A central problem at every SvenskMud-meeting I have attended is how to *gently* tweak the system technically so that the desired social effects appear. A variation of the social-technical design cycle, which I therefore think is more appropriate, is a model where the technical and social system interacts with each other, and where the technical system *contains* the social system (see figure 6.4 below). Mnookin (1996) comes

to the same conclusion regarding the relationship between the technical system and a specific part of the social system, namely the legal system within a mud:

“LambdaMOO’s petition process illustrates both the politics of technology and the technology of politics. [...] politics in LambdaMOO is implemented through technology and political conceptions can be embedded within the technological constructions of the virtual environment. In other words, ideas about politics can be, to a certain extent, hard-wired into society via technology.”

Mnookin, 1996a

The technical system *defines* the range of actions that are possible within the mud and so both enables and constrains the social system. Management is done through code.

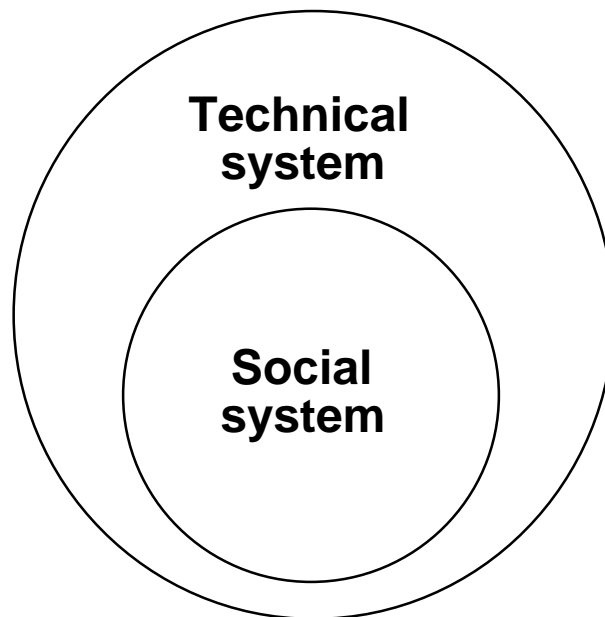


Figure 6.4. While the technical and the social system interact with each other, the social system inside a mud is ultimately dependent on the technical system, i.e. the computer program.

An example of *how a technical system defines the range of actions possible in a social system* is taken from Hippo Campus. That mud was established to teach first-year computer science students “about the social impact of computers and the ethical, legal, and social responsibilities of computing professionals” (Houle and Simon 1997). The reason why a university course about the social impact of computers is located inside a mud is because the social and the technical issues seamlessly fit together, thereby making non-technical issues considerable more appealing for the novice computer science students

“Within the Hippo Campus community, students find themselves dealing first-hand with issues [...] such as standards of conduct and antisocial behaviour, identity and authenticity [...]. They discover the tradeoffs between the legitimate need to monitor and the natural desire for privacy, and between system performance and enhanced security” [...]

Several technical projects entail examining and customising the environment. These are group projects, designed to enable the programmers and non-programmers in a group to make equal contributions to the work.

A major project concerns the formal establishment of a society within Hippo Campus. Groups of students consider such issues as personal freedom, codes of conduct, security-performance tradeoffs, access rights, censorship, privacy and confidentiality, voting, intellectual property, harassment, accountability, and enforcement in light of the capabilities and restrictions of the MOO. Each group then prepares a document outlining its position with respect to these issues, a proposed code of conduct and ethics, *a plan for its implementations* within the MOO and its anticipated technical ramifications.”

Houle and Simon, 1997, p.92-93, my emphasis.

An example from SvenskMud illustrates both how the technical and the social system are intimately coupled, and how changes in the technical system sets the frame for what is possible within the social system. The issue at hand was first discussed at the fifth SvenskMud meeting (1998). The largest and most ambitious city within SvenskMud is called Eriksros. However, the city did not contain enough hustle and bustle to live up to its size. At the meeting, it was decided that an attempt to make Eriksros more alive and livelier should be made. Some magicians volunteered to produce content that would fit in Eriksros, and it was decided that some other objects (for example a shop) should be moved to Eriksros from other places in the SvenskMud world.

Until the following meeting, six months later, Eriksros did grow. At that meeting, the sixth SvenskMud meeting (1999), it was decided that Eriksros should become the starting point and/or the natural meeting place in the game. As Eriksros grew a new problem arose. Even though Eriksros was now a bigger, better city, the players still did not come there. Instead they preferred to hang around the older but smaller city of Muddevalla and the players continued to do this even after the natural meeting place in Muddevalla was moved (i.e. a shop was moved). The players still hung around “discontinued” as the location of the ex-shop was come to be called.

At the seventh SvenskMud meeting six months later, an interesting discussion took place where the behavior of the players was analyzed. The conclusion was that it was actually not justified for players to pass Eriksros very often on their trips from one end of the SvenskMud world to the other, in terms of the SvenskMud geographical layout. The final outcome was therefore to move around some of the areas in SvenskMud, to close down some roads and open up some new roads. The goal was perhaps not to make “all roads lead to Eriksros”, but at least to make some of them lead there and so to make Eriksros into a more “natural” meeting place for the players. I suppose the goal was met since the subject was not raised at the following meeting.

Discussion

This is a chapter about software systems development. Despite the technical focus of this subject, one of the most salient features has been how technical and social issues are so intimately entwined that they finally become impossible to separate in a mud.

The chapter started with a list of nine characteristics of the systems development process in SvenskMud that, according to the traditional wisdom in the field of software engi-

neering should have doomed SvenskMud from the start. Obviously, something is amiss in the field of software engineering as the mere ongoing success of SvenskMud, Linux and other open source projects defy some of the “deep truths” that have been taken for granted in the field for decades²².

Parts of the answer that has been suggested in this chapter rely on technical solutions. Modularization in SvenskMud and elsewhere is a way to master complexity and this technique is relevant as part of the answer to several of the initial nine challenges. Another part of the answer is that SvenskMud works “good enough”, where good enough can be far from perfect in a technical sense²³. The full answer as to why SvenskMud survives and thrives in the face of these “insurmountable challenges” can not be given here. As of yet, no-one yet knows the ultimate answers as to *why* open source *actually* works, how long and how far it will work etc., only *that* it seems to work²⁴. The current state was not guessed by anyone ten years ago and no one paid attention to the success of open source until well after it was on its way.

Despite this, I will suggest some reasons *why* open source works and one reason why it works *now*. As has been suggested in this chapter, the key to understand the open source model of developing software is nothing less than a totally new way of thinking about software, about software development as a process, about software bugs and debugging, about developers and users and finally about community. This thinking shifts the frame from a traditional engineering perspective of *building* programs to a more biologically influenced perspective of evolving or *growing* programs incrementally (Brooks 1975/1995). This would place the emergence of the Linux/open source model of software development into a greater movement of looking toward biology for models on how to understand complex phenomena and how to construct or *evolve* complex artifacts²⁵ (Simon 1969, Langton 1989, Langton et al. 1992, Kelly 1994, Resnick 1995, Epstein & Axtell 1996).

SvenskMud as a system is very much “evolved” rather than “engineered”. Few attributes of a traditional engineering perspective are present as there is no blueprint, no central agenda, no common problem or goal that SvenskMud attempts to attain²⁶, no software architect or chief programmer who has a grand plan in his mind and makes all the important policy decisions etc. Furthermore there are no absolute rules for what “fits” in SvenskMud and what does not, and the guidance that does exist is very loose, permissive and negotiable. The individual magician has major freedom in deciding what to create. In deciding his own tasks.

The drawback is the conceptual disunity within both “SvenskMud – the game” and “SvenskMud – the computer program”. The metaphor of a “great babbling bazaar of differing agendas and approaches” is even more apt for SvenskMud than for Linux for two

22 I will pretend that SvenskMud is an open source project for the scope of this discussion because the differences between SvenskMud and open source do not make a difference to the basic arguments presented here.

23 It is beyond the scope of this work to judge the actual quality of the SvenskMud computer code.

24 The most plausible and initiated theories on the success of open source are those of Raymond that he develops in three papers; “The cathedral and the bazaar”, “Homesteading the Noosphere” and “The magic cauldron”. All three can be found in Raymond (1999).

25 There are interesting connections between, on the one hand building versus growing programs and, on the other hand, a mechanically contrived *Gesellschaft* (society) and an organically grown *Gemeinschaft* (community) (see chapter 1).

26 Two official goals of SvenskMud have been formulated (see chapter 4), but both goals are very general and none of them provides any real guidance for how to go about to design either “SvenskMud – the game” or “SvenskMud – the computer program”.

different reasons. *The first* reason is that Linux is an operating system and as such has a very specific and instrumental purpose. SvenskMud's purpose is, on the other hand, much less clear and this fact gives individual magician-programmers more leeway to develop SvenskMud in any direction that he or she sees fit. *The second* reason is that the concept of different agendas and approaches are applicable not only to SvenskMud's software development process, but also to the very content of the SvenskMud world itself and to the individual players' agendas and approaches. The scope of different agents is very wide in an open-ended system such as SvenskMud.

The shift from building to growing programs implies that we are talking about a shift in attitudes and approaches. This shift is important but it can only be part of an answer of why open source works. Another part of the answer is to answer the question of why it works *now* (i.e. instead of 10, 20 or 30 years ago)? My answer is that the proper structural conditions exist now but did not do so earlier. In order to illustrate this argument, I will reuse two quotes from earlier in this chapter. Agresti (1986, p.4) wrote that "because computers were an expensive resource, it made sense in 1970 that access to them should be preceded by careful planning so that time on the machine would be used effectively". Interestingly, and also very pertinent in this context, is the fact that 1970 is one of the last years when there were more programmers than computers in the world (Musa, 1983). The second quote is the definition of open source by Dyson (1998), where "open source is basically software developed by uncoordinated but collaborating programmers, using freely distributed source code and the communication facilities of the Net".

What these two quotes indicate together is the extent to which computers have entered our every-day lives and the fabric of society during the last 30 years. I will point out three differences that make a difference between the present and the past. *The first* is the extremely limited access to extremely expensive computers in 1970, and the careful planning that proceeded each rendezvous with a computer. This is the very anti-thesis of the free-wheeling tinkering with inexpensive home computers by hundreds of thousands of enthusiasts today. SvenskMud, Linux and other open source software are dependent on these people and their now-liberal access to computing power. To summarize, access to equipment and opportunities have changed radically.

The second difference that makes a difference is the "communication facilities of the Net". Not only have hardware costs plummeted, allowing many to own their own home computers, but there is also the Internet to tie everyone and everything together. Several commentators have pointed at the Internet as *the* critical success factor for open source. The Internet serves several functions and the most important for the development of Linux is that it uses the entire Internet as the potential user/developer base. SvenskMud uses the entire Swedish-speaking population who likes computer games, the fantasy genre and have Internet access as its potential user/developers base²⁷.

The third and final difference that makes a difference are the people who make it happen. Linus Thorvalds was a university student with his own personal computer and with Internet access during the very earliest years of the 1990's, and, he had been programming computers since he was ten years old. Linus Thorvalds was early both as to come around to programming computers and as to having access to the Internet. However, during the second half of the 1990's, a whole generation that fulfills two important criteria has come

²⁷ This population is of course quite a bit smaller than the potential Linux user/developer base, but it is in fact still not *that* small. The description fits many young people in Sweden.

to maturity. The first criterion is that they are young (often university or high school students or younger professionals) and have a lot of time on their hands comparatively. The second criterion is that they have had access to home computers for 5-10 years or longer, they have 5-10 years of experience of programming computers and they have Internet access today.

Besides these technical/structural reasons above that help explain the success of open source and SvenskMud in the 1990's, there is also an essential social dimension behind why SvenskMud, as a computer program and as a software development process, works. Let us reexamine Raymond's lesson number four, "If you have the right attitude, interesting problems will find you". That lesson might have seemed a little out of place in Raymond's list of lessons, not the least because Raymond gave no explication of what it really meant. It is not out of place. The lesson is of outermost importance and it does Raymond credit that he included it. However, Raymond does not have the conceptual structures to fit it in and to make it make sense in the context of his list. The lesson clearly has nothing to do with proficiency or technical abilities, with how good a person is at programming. It definitely has nothing to do with education of formal training. It does not even have any direct relevance to software systems development. However, it has all the more to do with socialization and community. *Having the right attitude* has to do with meaning, doing, experiencing and becoming (identity). It also has to do with picking up, assimilating and appropriating the shared values, goals, concerns, routines, procedures, practices, symbols and artifacts of the community in question. Not only will interesting problems find the person with the right attitude, but that person will, after the problem has been found, also know how to behave in order to solve it in the – according to the community – appropriate ways.

There are numerous hints in this chapter as to the importance of community and how shared values etc. are created and propagated in SvenskMud. This is evident in SvenskMud in the repeated emphasis on the value of cooperating with other magicians. It is evident in the creation of a shared history through the practice of only upgrading but never throwing away old code. I go as far as to state that a strong, shared set of values etc. is the very reason why SvenskMud works, despite the seemingly bad odds and lack of blueprints, central agenda, project leader, detailed formalized goals and rules etc. It is the *presence of community* that makes the difference.