Open-source software development methodology

LUDVIG A. NORIN (Ludvig@home.se)
FREDRIK STÖCKEL (Stockel@home.se)
Open-source software development methodology

Submitted to Luleå university of technology in partial fulfillment of the requirements for the degree of Master of Science in System Development and Software Engineering (240 ECTS Credit points).

by

Ludvig A. Norin
Fredrik Stöckel

Luleå, Sweden, 1998-05-20

Supervisor:
Guohua Bai, IES/DSV, Luleå university of technology
This paper is the result of a partial fulfillment of the requirements for the degree of Master of Science in System Development and Software Engineering (240 ECTS Credit points) at Luleå university of technology in Sweden.

This paper has been written with the kind help of several people, and we can’t really mention them all here. There are, however, a few people we would like to send special thank’s to:

Örjan Lundberg (lundberg@home.se) for useful material supplied at odd times.

Erik Lavander (erik.lavander@home.se) for important comments on the paper.

Rick Moen (rick@hugin.imat.com) for extensive comments on grammar and spelling.

Andrew Wu (wu@li.net) for the very helpful feedback.

Tom Werges (tom@citiscape.com) for good comments on the overall language.

Vernon Wells (wells@madhouse.org) for feedback in great spirit.

Rob Malda (malda@slashdot.org) for publishing our announcement of the paper at the famous and overall incredible news site SlashDot (http://slashdot.org). After the announcement we received over one-thousand unique hits on the page of the announcement.

And, most of them all, we want to thank Guohua Bai, our supervisor in the writing of this paper.

We wish you all the best of luck in the future.

Luleå, May 20th, 1998

_________________________ _________________________
Ludvig A. Norin Fredrik Stöckel
Free software is gaining popularity, and media attention. Open-source software is a kind of free software that is distributed not only in binary, but also in source code form. As there are lots of successful open-source software development projects, in terms of quality as well as popularity, it feels natural to ask certain questions on how these development projects actually works.

This paper takes on an explorative approach to open-source software development methodology, examining several projects and analyzing them in context of widely known software development methodologies such as life-cycle, prototyping and the spiral model. The authors argue that these projects don’t fit those methodologies, and that a model describing version-management oriented development is a good description of the methodology used in open-source projects.
# Table of contents

1. Introduction ................................................................. 1  
   Introduction ................................................................ 1  

2. Purpose/method/scope .................................................... 3  
   Purpose ......................................................................... 3  
   Hypothesis .................................................................... 3  
   Scope of study ................................................................ 3  
   Paper outline .................................................................. 3  
   Scientific method .......................................................... 3  
   Target audience ............................................................. 5  

3. Background ......................................................................... 7  
   Software for free ............................................................ 7  
   Copyright vs Copyleft ....................................................... 7  
   Open-source ................................................................... 8  
   The insider’s view ........................................................... 9  
   Engineering software ..................................................... 10  
   Summary ....................................................................... 10  

4. Engineering open-source software .................................... 11  
   The K Desktop Environment ............................................. 11  
   Mozilla the web browser .................................................. 12  
   Other open-source software ............................................. 12  
   Project management and organization ............................... 13  
   The developers and the users .......................................... 13  
   Will and duty .................................................................. 14  
   Summary ....................................................................... 14  

5. Software engineering methodology .................................. 15  
   Engineering software ..................................................... 15  
   The life-cycle paradigm ................................................... 15  
   The prototyping paradigm .............................................. 16  
   The spiral model paradigm .............................................. 18  
   Summary ....................................................................... 19  

6. Identifying the methodology ............................................ 21  
   The questions we ask ...................................................... 21  
   The paradigms and open-source software development ...... 21  
   Evolutionary software development .................................. 22  
   Summary ....................................................................... 23  

7. Finish ............................................................................. 25
1. Introduction

Introduction

What is our view of software development methodology? Is the development project of a software product limited to the employees of a successful software company? Are there alternative ways in which we may create software that is of high quality and satisfy our demands of usability and function? In this paper we explore and try to identify the methodology used in open-source software development projects.

Users of a software system are focused on the usability and function of a software system, but developers also focus on how the software system was actually developed. The discussions of software development methodology today is usually limited to in-house software development, and development of off-shelf commercial products. Not very much is written about freely available products, that are developed by a community of highly-skilled and driven hackers on the Internet. Is it so, that we believe these products generally exhibit lower quality, or in some other way is inferior to possible commercial alternatives?

The Internet itself is to a large extent run by non-commercial, free software products such as Bind, Sendmail and Apache. These open-source software products are also continuously developed and maintained by users of the Internet -- but by whom exactly, and why? And exactly how is it done? May it be so, that the software development methodologies taught in each and every computer science course are applied in these projects, or are they executed in an completely ad-hoc, unpredictable manner?
2. Purpose/method/scope

Purpose

The purpose of this paper is to identify the development methodology used in open-source projects.

Hypothesis

The development methodology used in open-source software projects is similar, in parts, to widely known software development methods.

Scope of study

This study deals with the methodology used in open-source software development projects, and not with specific managerial aspects such as project or risk management, except to the extent such aspects are covered by the widely known methodologies we use to analyze the open-source projects in our case study.

Open-source projects are mainly an Internet phenomenon -- but certainly not limited to the Internet as a communications medium. In this study, we focus only on open-source projects developed on the Internet.

When we use the word *methodology*, we refer to the overall development process and not the specific actions taken in a specific software development project. Therefore, we are not trying to document each and every step of development in the projects we have studied, but rather are trying to give an overall view of how these projects evolve.

Paper outline

As the concepts of open-source software may be unknown to most of this paper’s readers, we have divided this paper into three parts. The first part (chapter 3) is an extensive background on open-source and free software terminology. The second part (chapter 4-6) is a presentation of our case study, an introduction to three software development methodologies, and an analysis interconnecting the case study with those methodologies. Then follows a presentation of our conclusions (chapter 7) and a short epilogue (chapter 8) discussing various aspects of this paper.

Scientific method

As the purpose of this paper is to identify the methodology used in open-source software
development projects, using widely known software development methodologies as a starting point for the analysis, we have chosen to use an explorative approach in which we enhance our understanding of the problem domain while building the theoretical framework necessary for the analysis. This iterative process has the overall characteristics of the hermeneutic circle [THU91]. We have divided the process into three steps (figure 1), and begun by getting acquainted to the problem domain, defining overall concepts and choosing a suitable level of abstraction for our analysis. This first phase was characterized by studies of works written by, among others, Pressman, Sommerville, Mathiasen et al., Budde et al. and Raymond [PRE92, SOM92, MAT95, BUD92, RAY98a].

In the second phase, we chose two open-source software development projects of which we would conduct a deep study: the Mozilla and KDE projects. We also chose eight other open-source projects, which we studied only in their most obvious aspects: the Sendmail, Fetchmail, GIMP, GNUstep, GNU CC, EGCS, Gnome and GTK projects. These projects were chosen on basis of age, success and size, and we strived to get a widespread distribution of different sized projects.

The studies of these projects have been done in an explorative manner, using available resources on the Internet in which these projects are documented, as well as by participating in the Mozilla and KDE projects.

In our approach to the problem domain, we have tried to keep the level of abstraction as low as possible. The level of abstraction is then raised as we gained better understanding of the problem domain through literature studies and analysis of the case studies. The overall characteristics of this iterative process are reflected in the explorative spirit of this paper.

In the final, third phase, we make some general conclusions based on the analysis made in the previous phase.
Target audience

The target audience of this paper is anyone with an interest in software engineering methodology in general, and in open-source software development in particular. Most people with only a general understanding of software engineering should be able to grasp the discussion made in this paper, although a thorough background in software engineering will make it more accessible. If the reader has also been a contributor to open-source projects this paper may provide additional value, in that it strives to present a new approach to the understanding of such projects.
3. Background

“...Spacewar was not sold. Like any other program, it was placed in the drawer for anyone to access, look at, and rewrite as they saw fit. The group effort that stage by stage had improved the program could have stood for an argument for the Hacker Ethic: an urge to get inside the workings of the thing and make it better had led to measurable improvement. And of course it was all a huge amount of fun.”

-- Steven Levy [LEV94]

Software for free

Ever since the very first computers were manufactured, there has existed software that could be obtained for free. In the early years of the computer industry groundbreaking work was done in programming, which was not only available free of charge, but even distributed free of charge. The concept of sharing research data that is commonly encountered in the academic world is in many ways analogous to the way computer programs were shared amongst users in the early sixties and seventies, and even today, [AXE95, LEV94]

To place software in “the public domain” is a common way of giving away software for free, without imposing any restrictions on either use nor distribution. Software placed in the public domain may be used by anyone, in any imaginable way; it is not covered by copyrights or license agreements [STA98]. Today, many software companies release software free of charge to the public (for various reasons), but they rarely place the software in the public domain -- instead, they use license agreements, copyrights and patents to restrict the allowed usage of their work, although it is available free of charge. This is also common amongst individuals and non-profit organizations, but the very purpose of the restrictions imposed on the software differs greatly.

Expressions such as “free software”, “open-source software”, “shareware” and “freeware” are commonly used in reference to software that is given away for free, and, for the discussion presented in this paper, we will need a clear distinction among the different kinds of such software.

Copyright vs Copyleft

In order to separate different kinds of copyright restrictions, the term copyleft was introduced. The GNU\(^1\) project, founded by Prof. Richard M. Stallman, summarize the term like this:

---

1. GNU (GNU is Not Unix) is a project that aims on creating a completely free Unix-compatible software system. As of today this project has produced, among lots of other things, widely used compilers and debuggers such as GCC and GDB. The project is run by the Free Software Foundation (http://www.fsf.org).
“Copyleft says that anyone who redistributes the software, with or without changes, must pass along the freedom to further copy and change it. Copyleft guarantees that every user has freedom.”

-- Richard M. Stallman [STA98a]

In contrast to the restrictions usually imposed on software using copyrights, which are aimed on restricting distribution and usage, the copyleft principle means applying restrictions to the software in order to ensure that any copies of the program will always be legally distributable free of charge. The copyleft also ensure the software may be legally modified and incorporated in other software packages. This is done by applying a licence to the software, which defines the rules under which it may be distributed.

Software placed in the public domain is non-copyrighted, thus also being non-copylefted. Free software distributed by commercial interests are usually copyrighted, but not copylefted, in order to preserve rights of usage, distribution and modification to the author or owner. However, much of the free software that is distributed by non-commercial interests are copylefted, and distributed not only as binary packages, but also in source code form. The principle of copyleft ensures that any modifications made to the software does not affect the rules under which the software may be used, distributed or modified.

In any discussion about free software it is necessary to make the distinction between copyright and copyleft, as the word ‘free’ may be interpreted as either free of charge, or in the sense of freedom [STA98c]. This paper is written upon the assumption that free software is synonymous to copylefted software, thus not necessarily being free of charge, but always being free to use, modify and distribute.

The set of rules defined by copyleft licences are designed in such way as to permit any changes to the source code, but no changes of the licence terms under which the software is distributed. This way any modified versions of the software has to be made available for anyone who wants it: Many copyleft licences make exceptions for private usage of the modified source, though.

Open-source

Software distributed under such licence terms as copyleft is always distributed in source-code form and often, but not always, in binary form. Most copyleft licences are designed to not only permit modifications, but also to encourage people to help the code evolve. It is such licences we refer to when using the expression open-source software.

The idea of open-source software development is to allow and encourage people on the Internet to help a software project evolve. This form of software development has many advantages -- for example, a company may create some kind of utility to solve problems experienced in a network administration which can’t be solved in any standard way. Provided the software is of interest to other organizations and individuals, it will evolve -- bugs being corrected and new features being added, without great effort by the company
The ‘open-source’ expression was introduced in early 1998 as a less confrontational way of referring to free software. It has been argued that the word ‘free’ is detraacting commercial interests from adopting the idea of open software development, as the word ‘free’ often is interpreted as ‘free of charge’ rather than in the sense of ‘freedom’. Since the introduction the expression has been adopted by media and developers, and even by major software companies such as Netscape Communications, Corp. and Corel Computer Corp.

In the middle of 1997 a paper called “The Cathedral and the Bazaar” was published by Eric S. Raymond, as an effort to summarize the development styles used in open-source software development. Although he successfully identifies and explains two different development styles, no reference is made to any widely known development methodologies such as waterfall, the spiral model or prototyping. This paper was conceived in order to explore the open-source software development methodology, based on widely known development methodologies.

The insider's view

An open-source project may take various forms. Raymond identify two development styles and consider them dominating the open-source development community:

- **The Cathedral style of software development**
  Software developed in the cathedral style is typically crafted by a single programmer, or a small isolated group of programmers. Although contributions of source code may be accepted from people not associated with the project, it is not the primary development style. Design and implementation decisions are exclusively made by the authors, as are the release cycle. In many cases there are very few releases of the software until it is considered usable for a larger user community, thus mailing lists making it harder for programmers to contribute code to the project.

- **The Bazaar style of software development**
  Projects executed in bazaar style basically adheres to the rule “release early -- release often”. Although many projects are run by a small group of programmers, the software is released as often as possible to encourage contributions from other programmers. Design decisions are often discussed intensely on mailing lists, newsgroups and IRC. Often there is only one central code repository, or source tree, to which contributions are added. When developers disagree on implementation techniques the source tree may split, as the repository manager may chose to incorporate only one of the proposals into the source tree. This way the original project may deviate into several related but rather different projects, even though this is quite uncommon.

These fundamental development styles are very useful in describing the how an open-source development project is managed, but tell us very little about the specifics of the analysis, specification and design phases found in most software engineering methods.
Engineering software

Software development projects usually obey some kind of structure, following given timelines and entering different stages during the development process -- such as analysis, specification, design and implementation phases. The use of a software engineering methodology is fundamental to most application developers today. The methodology is an important tool to control such things as resource management, product design and quality assurance, in order to produce "well-engineered software". [PRE92, SOM92]

Sommerville [SOM92] argues that a well-engineered software system possesses four key attributes, assuming the software provides the required functionality:

- **Maintainability**
  A software with long life-time is subject to regular change, and has to be written and documented as to make changes possible to the code without undue costs.

- **Reliability**
  A software should perform as expected by users, and not fail more often than is allowed for in its specification.

- **Efficiency**
  A software should not make wasteful use of system resources such as memory and CPU.

- **An appropriate user interface**
  To allow users take full advantage of a software system, the user interface has to be designed with the capabilities and background of the users in mind.

Different software development methodologies take on different approaches in order to satisfy these attributes -- in particular users are given different roles in the development process and the design and requirement specifications are made in different ways.

Summary

Software development methodology is essential to the software engineering process, and even though we have a useful description of the fundamentals of the open-source development process, there is a lack of descriptions of the methodology used in open-source development. Let us now take a closer look at some different open-source projects, to learn a little about how they strive to achieve the attributes that are essential to well engineered software.
4. Engineering open-source software

“*I consider that the golden rule requires that if I like a program I must share it with other people who like it.*”

-- Richard M. Stallman

The K Desktop Environment

For many years there has been a lack of useful graphical environments for Unix systems. The framework for graphical user interfaces found in most Unix systems, X/Windows, is powerful, but does not pose an attractive user interface for end-users. For this purpose window-managers were introduced in order to help users take full advantage of the graphical windowing environment. Standard libraries for application appearance such as OSF Motif and XView were also introduced, and sometimes were made a more or less integral part of the window manager. Yet there was a lack of a desktop metaphor as found in Apple MacOS and Microsoft Windows systems for a long time, until major vendors agreed on a standard for such integrated user environments, called the Common Desktop Environment, which today is a dominant in commercial Unix systems.

With the exceptional growth of users of non-commercial, free Unix-like operating systems such as Linux and FreeBSD, a need for a free implementation of a graphical user environment similar to the CDE emerged. The K Desktop Environment, KDE, is a project initiated in order to satisfy that need.

The first public beta release was announced one year after the project was initiated, and in the following seven months there were three additional public beta releases. Distribution of the source code began long before the first public beta was released, using a client-server based version control system which supports remote collaboration and multiple developers. After a year and a half more than two-hundred individuals had made different kinds of contributions to the project.

Anyone is invited to participate in the project; design and implementation strategies is discussed on mailinglists and it is common that people implements a feature, making it available by other means than the version control system before it is incorporated into the main source-tree. Great efforts are also put into documentation and internationalization work. One of the design goals is to provide developers with a powerful source code library to be used in creating applications for KDE -- a goal that has been proven very successful. The library is important not only in order to make it easy to create GUI applications, but also to make those applications consistent in their look and feel. Guidelines were set up at an early stage in the development process as means of encouraging application developers to give their applications certain features such as on-line help and coherent access to application functions.

Each public beta release has been preceded by a short test period, in which only error corrections has been accepted in the main source-tree. At the end of the test period the KDE packages are published at more than twenty FTP sites around the world, and
announcements are made on various WWW sites as well as mailinglists.

Mozilla the web browser

Mozilla is the generic name of web-browsers derived from the source code of Netscape Communicator, which Netscape Communications Corp. released at the end of March 1998. When Netscape chose to turn the Communicator product into a open-source development project, the goal was to increase the development pace by incorporating contributions from the many developers present on the Internet, as well as to seed market for Netscape’s enterprise solutions.

When the source code was released the product development continued within Netscape, and a small group of people were assigned to coordinate the development project, called the Mozilla organization. The role of this organization is to provide directions for further improvements to the software and to integrate contributions into the source-tree, and also to help open-source developers reach consensus in order to coordinate development. The Mozilla organization provides access to the source code by means of a version control system, and by releasing code snapshots on a regular basis.

As the Communicator product is still being developed by Netscape, the different teams of programmers now act as contributors to the Mozilla organization in precisely the same manner as any other contributor. In this arrangement the Mozilla organization acts as a provider of source code to, among others, Netscape so that they may release their Communicator product.

The Mozilla organization provides communication channels for developers such as IRC, newsgroups and mailinglists. Design decisions are discussed, drafted and decided on cooperatively by all participants on those channels, and published on the Mozilla organization website. The website also provides coding guidelines, source code roadmaps and lists of works in progress as well as a bug tracking system.

Other open-source software

The Free Software Foundation is making an effort to create a free implementation of the OpenStep specification as published by NeXT Software Inc. and SunSoft Inc. in 1994. This project, called GNUstep, is divided into several sub-projects which are managed rather independently from each other. Of particular note is that these projects didn’t produce any code at the early stages, creating design specifications and requirements instead. When the coding started very few developers did a majority of the work, and even though the project begun in 1994 the first public beta was released in 1998.

In 1979 the program now known as Sendmail was written by Eric Allman at the University of California at Berkeley. Sendmail is as of 1998 the dominating electronic mail routing implementation, and has received major contributions not only from individuals on the Internet, but also by Sun Microsystems Inc. and Hewlett-Packard Company. The product evolves continuously mainly in maintenance aspects such as bugfixes, but new
Other free development efforts have split up, creating new applications. The compiler created by the Free Software Foundation called GCC has spawned a new project called EGCS, executed by Cygnus Solutions, adding features not incorporated in the original source-tree. Another project aiming to create a high-end image processing application called GIMP has spawned an effort to create a user interface library in parts similar to the OSF Motif library -- and that effort is in turn the basis of the GNOME project, which like the KDE project is developing an integrated user environment with attractive developer support libraries.

Project management and organization

Of vital importance to all mentioned development projects is communication between the contributors. This is accomplished in many different ways, and often there are no single one preferred communication channel in a open-source project. The use of newsgroups and mailinglist dominates, and the web is almost always used as a tool to create a central source of information.

Distribution of source code is done in various ways. Many projects use version control systems, of which Concurrent Versions System, CVS, is dominating. A common way of distribution of source-code to developers is to allow anonymous access to the source-tree using a CVS client. It is also common to place snapshots from such a CVS source-tree on FTP sites that are mirrored around the world. The version control systems are fundamental to the project management of these projects, as it aids in keeping track of changes in the source-tree, making it possible to reverse changes done to the software. In most cases it is easy for developers to gain full access to the CVS server, and the contributions made by those who only have anonymous, read-only access are sent to those who have full access.

Some of the projects we have examined, such as the GNU C compiler project are more centralized than others. Although all projects utilize some kind of central coordinating organization, the influence this organization on the contributions to the actual software differs.

The developers and the users

The users of open-source software are not always developers. There is, for example, a vast number of Sendmail and Linux users that are not software developers. Many projects actually strives to widen this “ordinary” user-base, although a software developer also is a potential contributor to the project. This is because most users in fact are co-developers of the project, helping it evolve in other ways than contributing code. An example of this is the process of bug-correction often encountered in these projects. It is common that bugs are reported, identified in the source code and corrected by different individuals. The “ordinary” users that have no programming experience are often prone to report bugs, and as there are many users that are also programmers, chances are that at
least one programmer takes the time to identify the error in the actual source code -- and as the error is identified someone else often takes time to correct it.

Other kinds of co-development efforts, such as introducing ideas for new features, translating a program to another language and writing documentation is often done by users that are not programmers. These tasks is almost always done incrementally and given a large user-base, much work in these areas will be done by many individuals making quite small contributions.

Will and duty

The software projects we have studied evolves for various reasons. Mozilla the web-browser has strong commercial interests behind it while the people involved in the KDE project simply exhibit a strong desire to succeed. As an open-source project grows, gaining a bigger user-base, there will be greater need for the software to evolve. As the need grows, so does the pressure on the core developers in the project. In this situation one of two things usually happens:

- The project is split up into two or more smaller projects
- The number of developers that make very large contributions increase

The latter of these is very much due to the fact that many users are developers, and as most users are aware of that there are no guarantees given by anyone that the software will evolve at a certain rate, one has to give in order to gain. The first is a direct consequence of the principle of copyleft -- instead of switching to a new software package, a group of skilled enthusiasts are legally allowed to reuse parts of the original code and extend it in order to satisfy the needs that are not met by the original development project.

Summary

The projects we have studied all take on an evolving approach to the software development process, a fact that is largely due to the influence of the copyleft licence which both allows and encourages people not associated with a project to work on the source-code. There are, however, some differences in the centralization of development efforts in the projects. We have noticed that the projects in which the developers are a small, centralized group there are longer release cycles, and less encouragement of contributions from people not part of the core group of developers.

In the next chapter we are going to find out a little bit about some widely known software development methodologies, to see if there are great similarities or differences between the methodology described in those methods and the way these projects are executed.
5. Software engineering methodology

“Given unlimited resources, the majority of software problems can probably be solved but the challenge for software engineers is to produce high-quality software with a finite amount of resources and to a predicted schedule.”

-- Ian Sommerville [SOM92]

Engineering software

Developing a software system is usually not done in a single night of hard work, but is a complex and time-consuming process. In order to control this process, reducing the complexity and uncertainties surrounding the to-be software system, we try to adhere to some kind of framework that introduces certain degrees of structure to the overall development process.

Software engineering methodologies are the framework that tells us how we should go about developing our software systems. These frameworks define different phases of the development process, such as planning, requirements analysis, design, testing and maintenance. [PRE92, SOM92]

The most popular methodologies for software engineering are sometimes referred to as software engineering paradigms. The choice of which methodology to use in a development project is closely related to the size of the software system and the environment it is supposed to function in. The environment in itself constitutes a larger system -- though most often not a computer software system, but rather some kind of organization. The different paradigms presented in this chapter all spring from this view of the world as a system of systems.

The life-cycle paradigm

The life-cycle paradigm of software engineering is sometimes called the “waterfall model”, as it demands a sequential approach to the development process. The work is started at the system level and passes through phases of analysis, design, coding, testing and maintenance. Six activities constitutes the overall development process (figure 2, next page). [MAT95, PRE92, SOM92]

- **System engineering and analysis**
  This activity is characterized by system-level requirements gathering for all system elements, which then is reduced to a subset of requirements that are relevant to the software system being developed. Overall design and analysis tasks are also executed in order to understand the full system of which the software system being developed is only a part.

- **Software requirements analysis**
  This activity is usually executed together with the customer, as the goal is to docu-
ment all function, performance and interfacing requirements for the software.

• **Design**
  When creating the design of the software system the requirements are transformed into a representation of software that can be assessed for quality before the actual coding begins. Data structures, architecture, procedural detail and interface characterization are outlined and documented in a design specification.

• **Coding**
  This activity is the transition of the design specification into a software program.

• **Testing**
  This activity is to be executed using documented test methods in order to ensure that as many errors as possible is unveiled, and that the software is in accordance with the requirements.

• **Maintenance**
  Rather than being an atomic activity, maintenance reapplyes all other activities, as new requirements are defined in order to adapt the software to the changes occurring in its environment and to correct errors encountered after the software has been deployed.

**The prototyping paradigm**

Contrary to the static, procedural approach offered by the life-cycle paradigm, the prototyping paradigm can be used. Similarly to the life-cycle paradigm, this process begins by gathering requirements of the system (figure 3, next page). The developers meet with customers, determines the overall objectives of the software and identifies any known requirements. A quick design then occurs, focusing on areas visible to the users, such as user interface and basic functionality. The design model is then used to implement a first prototype, which may take one of three forms: [BUD92, MAT95, PRE92, SOM92]
• **Interactions prototype**
  This is a paper or computer software prototype, that makes it possible for users to understand how to interact with the software system.

• **Subset function prototype**
  This is a working software program that implements a subset of the required functionality.

• **Existing program**
  This is an existing program that implements most or all of the required functionality, but has features that should be improved in a later development effort.

When the prototype is created, it is reviewed by the customer. Typically this review gives feedback to the developers that helps remove uncertainties in the requirements of the software system, and starts an iteration of refinement in order to further clarify requirements by improving the prototype, or by building new prototypes. This process will result in one of two kinds of complete prototypes [PRE92]:

• **A throw-away prototype**
  This is may be a usable software program, but is not suitable as the final software product, for various reasons such as poor performance, maintainability or overall quality.

• **A prototype to refine and deliver**
  This kind of prototype is enhanced and possibly reworked in various areas so that it is suitable to deliver as the final software product.

It should be noted that there are generally more reasons to throw away a prototype than there are not to -- Brooks expresses this view like this:
“The question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers...”

-- Frederick Brooks [BRO75]

The basis of this argument is that the first system built usually is too slow, too big or too awkward in use -- and that the prototype serves a better purpose of refining requirements than refining code.

The spiral model paradigm

In addition to some of the aspects of the life-cycle and prototyping paradigms, the spiral model adds an element of risk analysis to the development process. The model is presented as a spiral (figure 4), in which each iteration is represented by a circuit around four major activities: [BOE88, MAT95, PRE92, SOM92]

- **Planning**
  
  Determine objectives and constraints of the project, and define the alternatives.

- **Risk analysis**
  
  Analysis of alternatives, and identification/resolution of risks.

- **Engineering**
  
  Development of the “next-level” product.

- **Customer evaluation**
  
  Evaluation of the product engineered.

In each iteration the requirements are refined and a more complete prototype is produced, either by building on the prototype created in the first iteration, or by creating a
new one. The risk analysis always ends in an no-go decision, at which the project may be terminated if risks are considered too great. The engineering that is done in each iteration may be executed in life-cycle form as well as prototyping, depending on the certainty of the requirements. \[\text{PRE92}\]

As the developer and the customer are given an opportunity to react to risks at each iteration in the model, it can be considered to be \textit{evolutionary}. It allows the developer to use prototyping approach at any stage in the development, still maintaining the stepwise systematic approach of the life-cycle paradigm.

\section*{Summary}

Regardless of project size, complexity or application area there are three generic phases that always occur in some form in these paradigms, namely \textit{definition}, \textit{development} and \textit{maintenance} \[\text{PRE92}\]. In the definition phase some kind of \textit{system analysis} is performed, the project is \textit{planned} and to some extent \textit{requirements} are defined. In the development phase some kind of \textit{design}, \textit{coding} and testing always occur. Finally, the maintenance phase is characterized by \textit{correction}, \textit{adaptation} and \textit{enhancement} activities.

We now move forward with an analysis of the open-source projects presented in chapter 4, based on the characteristics of the paradigms presented in this chapter.
6. Identifying the methodology

“What sometimes turn out to be blind alleys are often presented as super-highways leading to the future.”

-- Andrew L. Friedman [FRI89]

The questions we ask

We have previously described how different open-source development projects are run, and explained three well-known software development paradigms. By applying this knowledge it will now be possible to analyze the open-source development methodology in order to verify our hypothesis -- that open-source software development methodology is “...similar, in parts, to widely known software development methods”.

Is it possible to relate the different open-source development projects to the development methodologies presented in the previous chapter? Or are the projects we studied fundamentally different to the kinds of projects that are targeted for those methodologies? In this chapter we try to figure out the answers to these questions.

The paradigms and open-source software development

The sequential approach demanded by the life-cycle paradigm is not consistent with any of the open-source projects we have studied. Though a few projects start out by creating design specifications and collecting requirements, the majority of the projects we examined quickly changed focus to implementation of the software system. The system-engineering phase is not applicable at all in these projects, as there are no single target organization for the software being built. Design and analysis phases are identifiable in the GNUstep project, but that project has not continued with coding and testing in a sequence similar to the life-cycle paradigm, but rather entered a parallel coding and testing phase.

Contrary to the life-cycle paradigm, prototyping seems more appropriate as description of the development methodology used in these projects. There are, however, an important aspect of the prototyping paradigm that is not consistent with any of the projects examined, namely the view of the prototype as being a tool for better understanding of customer requirements. Even though each new version released by any of these projects better reflects the requirements of the users, the versions are not usually released in order to be evaluated by the users. Rather, software is released in order to make new features, bug fixes and enhancements available for anyone as soon as possible after they have been incorporated [RAY98a]. The prototyping paradigm also makes a clear distinction of the end of the software projects, but none of the projects we have studied has reached a clear end of development, even though the oldest project goes back to the late seventies.

The spiral-model’s addition of an element of risk analysis is not applicable to the soft-
ware projects we have studied. In open-source projects anyone may continue any development already done. This is not consistent with one of the purposes of the risk analysis, which states that if the project is considered to be unsuccessful it may be abandoned. If the manager of an open-source project decides not to keep the project running, ending distribution of the source-code to developers, any other person may pick up the task of managing the source-code distribution. This has also happened in some projects though it is not very common, as distribution of source-code using for example the CVS versioning system is a task that doesn’t demand any great effort.

Obviously, there are great inconsistencies between the paradigms view of software development, and what is found in the open-source community. Does this mean that the work done in the open-source projects are chaotic, executed in a completely uncontrollable manner? To answer this question, we have to change the way we think about software development. Instead of looking at the mass of different individual open-source projects, which each uses its own way of management and control, we raise our view to that of the organization in which these softwares are developed -- the internet and the open-source community.

**Evolutionary software development**

Evolutionary software development is not a methodology in itself, but rather a view of software development. If we consider a software development project as a self-contained activity in an evolutionary perspective, it becomes evident that we are only considering a thin slice of the reality. The software development paradigms only consider the time period of actual construction of the software system, not taking into account the way in which the organization reacts to the software system, adapting work procedures and paths of communication. To this part of the development process the term *maintenance* is applied, and it is described as a never ending activity of improvements, corrections and adaptations -- that is known to account for approximately two thirds of the total technical effort required for the development process. [BUD92]

In the open-source projects we have examined there seems to be a never-ending process of software development, in which new features are added to the software and corrections are made. This process is not to be characterized as maintenance though, as the software always evolves to meet new requirements from the users. This is seen in all examined projects, and analogous to what may be referred to as versioning (figure 5, next page).

Budde et al. [BUD92] has identified a number of procedures that are commonly used in software development in which the benefits of evolutionary software development becomes apparent. One such procedure is versioning, a way in which many software firms market their products and adapt them to the wishes of their clients. Off-shelf software products such as database systems, spreadsheets and word processors are developed for a wide range of users in different organizational contexts. It is not conceivable to get an over-view of all requirements put by the users even by doing careful market analysis before creating the software system, as the different organizations inevitably will change, the users requirements will change, too. It is also not possible to incorporate the changes to a completed software system at all users sites -- instead the versioning approach is used.
The requirements for updated versions of software are collected by distributing pre-release versions of the software in development, by conducting public testing and by feedback from user community groups. The choice of whether to upgrade or not is done by the users, and this is possible mainly because they to greater extent are part of the development process.

In this model of evolutionary software development the software system will adapt to the changing requirements from the users, and in some cases never reach a maintenance phase. This is precisely what is happening in the open-source community, where anyone may not only give feedback to the software developers but also help develop the software themselves.

**Summary**

The software development paradigms presented in the previous chapter are not adequate descriptions of the open-source software development methodology, although there are similarities. The prototyping approach have similarities, but the open-source projects are not creating prototypes, but rather different versions of the software.

Instead, the open-source software projects are better described using an evolutionary approach to software development. In particular, versioning is useful to describe the way in which the software evolves, becoming more powerful in function and stability.

With this in mind, we now proceed to make some general conclusions about open-source software development methodology.
7. Finish

"We go about our daily lives understanding almost nothing of the world."
-- Carl Sagan [HAW88]

A few words on open-source

Open-source software development has gained a lot of attention recently, as Netscape Communications Corp. released the source code for their web-browser product, and as Corel Computer Corp. decided to use the open-source operating system Linux as basis for their network computer. It is sometimes stated that open-source software development is inconceivable in a commercial context -- as the source is free, how could you possibly make money out of it? This argument has been proved wrong by companies such as Caldera and Redhat that benefit mainly from open-source software efforts. It is yet to be seen if Netscape and Corel will add to those who do benefit from open-source. But first and foremost, it is the users that have benefitted from these free software products. To a large extent, the Internet itself is run by open-source software such as Bind and Sendmail, and many of the GNU Unix tools are taken for granted to exist on a decent Unix configuration. [RAY98a]

The open-source software development efforts that we have studied all have characteristics that makes them different, but as they all spring from the principles of copyleft licences they all share the common ground of evolving according to the same set of rules.

Not all software development projects are suitable to run in open-source form, for many reasons. It is, however, important to analyze these projects in order to better understand their place in the traditional view of software development -- because many projects may actually be suitable for open-source software development that one may not consider otherwise. As we now present the conclusions of this study, we hope to have taken a step in that direction.

Conclusions

The purpose of this paper was to identify the development methodology used in open-source software development projects, using widely known software development methodologies as a starting point in our analysis of the projects presented in our case study.

Open-source software development is not governed by the principles outlined in the lifecycle, prototyping or spiral model paradigms. The spring-point that makes these projects not fit those paradigms are that they don’t face the same kind of managerial and economic problems as traditional in-house software development projects -- as contributions to open-source software projects are done on basis of will rather than duty, they don’t adhere to traditional rules of pay-for-code development efforts.
The methodology used in open-source software development is best described using the theories of evolutionary software development, and particularly well defined using the versioning model presented by Budde et al. [BUD92], which illustrates the software system as an evolving entity in a system of user and developer communities.
8. Epilogue

A few words on reliability and validity

Why did we write this paper, on the subject of “open-source software development methodology”? Our interest stems from personal experiences of using open-source software, as well as from our academic background within systems science.

This paper is written in an explorative and descriptive manner, and it has as such been exciting, and writing it has been a learning experience. At times we’ve got a feeling of trespassing on unexplored fields -- and these are the times we feel have been most rewarding.

The discussion made in this paper is to a large extent aimed on finding and explaining the works of the open-source software development methods. As we have seen many positive aspects of this development process, we have come to believe that traditional development methods really may have something to learn from the open-source community.

What are then our experiences of our scientific approach? The source-material and ideas that is the ground upon which the paper is written have to a large extent been focused of traditional software engineering, and open-source software engineering. This material is very extensive, and have demanded a large portion of critical interpretation. We hope to have put forward the characteristics of open-source software development, in a way that have been natural and easy to read.

The reliability of the discussion, analysis, and conclusions of this paper is closely related to our results, and our interpretations towards the problem domain and the source theory material that is used in the analysis of the case studies. In order to ensure that our observations and analysis are as correct as possible, we have chosen to work very hard on understanding the problem domain, and what really is the relevant parts for us to study - as is also stipulated by the hermeneutic circle.

In reality, there are no irrelevant parts -- they are all significant for the whole of the development process, and the effects it will have on its environment, the organization. The limits of the scope of our study has felt both natural and necessary, in order to give us the possibility of exploring a smaller part of the open-source domain more thoroughly.

When discussing open-source, we find it hard to avoid subjects such as commercialism and idealism. When discussing open-source, we are not only discussing a methodology, principle or phenomenon -- open-source is in itself way of viewing software, and the governing principles of the development process. Commercialism and idealism found in the open-source community is in itself a subject well worthy a paper.

There are many other aspects of open-source software development that are interesting and deserve to be explored. We firmly believe that this is will be a subject of numerous future papers. As companies such as Netscape Communications Inc., and Corel Corp. are adopting the open-source way of software development, it certainly gains potential in becoming a successful business model for software development companies.
Further studies

• In what ways may commercial interests benefit of open-source software development methodology?

• Is open-source software development a feasible alternative to traditional off-shelf software development methods? What are the pros and cons?

• The way open-source software development projects are managed has evolved on the Internet, but is it possible to use it in in-house development projects? What is there to gain by doing so?

• To what extent is user participation a crucial for the success of an open-source software development effort?


