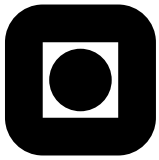
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK



FORDYPNINGSPROSJEKT

Kandidatens navn: Alf B. Lervåg

Fag: Datateknikk

Oppgavens tittel (norsk):

Oppgavens tittel (engelsk): Software Development Methodologies and Open Source Software

Oppgavens tekst:

Study existing models for COTS/OSS-based software development and find practices that the respective models can learn from each other.

Oppgaven gitt:	August 20, 2005
Besvarelsen leveres innen:	December 20, 2005
Besvarelsen levert:	December 20, 2005
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veileder:	Reidar Conradi

Trondheim, December 20, 2005

Reidar Conradi
Faglærer

Abstract

The paper documents the authors research into software development methodologies and how Open Source Software (OSS) projects can be seen in light of these methodologies. The paper continues with a brief look at what tools are typically used for open source project coordination, and concludes with a list of practices that are apparent in OSS projects but might not be as much used in non-OSS projects. Finally we suggest directions for further research.

Preface

This paper is the result of a literature study that was performed by a fifth year student at the Department of Computer and Information Sciences at the Norwegian University of Technology and Science (NTNU) as part of the course *TDT4735: Systemutvikling Fordypning*.

Acknowledgements

The author wishes to thank Reidar Conradi for being a patient and understanding advisor and for his help in deciding on the direction and focus of the study.

Truls Tangstad for valueable advice and conversations that were enlightening and helped further understanding of software developing as an art as well as a process.

Magni Onsoien for proof reading and helpful comments during the last few days of the development.

Contents

1 Introduction

The paper is an attempt to solve the assignment

> Study existing models for COTS/OSS-based software development and find practices that the repective models can learn from each other.

We have conducted a literature study of software development methodologies, and then tried to find aspects of Open Source Software (OSS) development that non-OSS projects can learn from, and vica versa. Our goal with this study is to get an overview of software development methods to build a foundation for further studies into this area.

Since this is a literature study, we have not discovered anything new. However, we have tried to find practices that can be extracted from the different methods and that can be used to improve existing methodologies.

We have also looked at tools that are much used in OSS projects and argue for why these tools can be used in non-OSS projects as well.

2 Development Methodologies

A development methodology is commonly refered to as a collection of ideas and practices for planning, designing and actually developing IT-systems.

McConnell claims that any approach to programming constitutes a methodology no matter how unconscious or primitive the approach is [McC04, p657]. He also claims that the point of most methodologies is to reduce communications problems (see Section 2.1).

While what McConnell says might very well be correct, we've decided to use the following definitions of development method and development methodology.

Method A codified set of practices that may be repeatedly carried out to produce software (e.g. the Waterfall model and XP). (Sometimes refered to as development models, e.g. the Waterfall model and the Spiral model.)

Methodology the interrelationships between software development methods (e.g. plan-driven and evolutionary methodologies). Sometimes refered to as life-cycle models [vV00, p48–49].

We now proceed with a discussion on formalism and then a presentation of different software development methodologies and methods.

2.1 Formalism

As the size of the development team increases, McConnell argues, the number of communication paths increases multiplicatively, proportionally to the square of the number of people [McC04, p650]. As the number of communication paths increases, so does the amount of time spent communicating and the risk of communication mistakes increases. Larger-sized projects need a way to streamline communication or limit it in a sensible way.

A typical approach to streamline communication is to formalize the communication in documents. Different development methods have different levels of formalism, as illustrated in Figure 1.

Formalism is also dependant on the criticality of the project, as argued by Cockburn [Coc02]. If the consequence of a failure in the program will lead to injuries or death then the required level of formalism is higher than if the result is a few hours of lost work.

2.2 Plan-driven development methodology

Plan-driven methods are based on the assumption that one can predict the requirements and design needs early in the development project, and that these do not

change during the project life time. If this assumption is true, then it should be much better to collect the requirements and plan a design that fits these requirements first, instead of having to redesign the software after the coding has begun because the initial design wasn't good enough.

Since experience has showed that it is very difficult to make perfect predictions, most plan-driven methods include some form for iterative development. This means that it is possible to learn something in one of the latter stages of the development and take this knowledge back to one of the earlier stages and redo or improve the work based on the new knowledge.

Plan-driven methods have a high level of formalism, usually in the form of documents for each phase of development. Because of this they are sometimes refered to as document driven methods. This makes plan-driven methods well suited for large teams and less vulnerable for changes in the project staff. Fowler argues that this also makes offshore development much easier [Fow04].

However, these documents are time consuming to create and maintain, something that is reflected in the term heavyweight that is also used to describe the methodology. (Quite fitting as they are slow to gain momentum, and difficult to stear after they have gained momentum.)

2.2.1 The Waterfall model

Royce [Roy70] presents the model though not under this name. It devides the software development process into 7 distinct phases that are sequentially dependant on each other, see Figure 2. Royce argues that there is a need for iterative interaction between the various phases and that these iterations should not be confined to the successive steps. This argument is based on Royces experience that predictions and planning often are wrong or incomplete and that when this is discovered at later phases there is a need to go back and fix the problem before moving on.

Much of the argumentation for using the Waterfall model (and plan-based

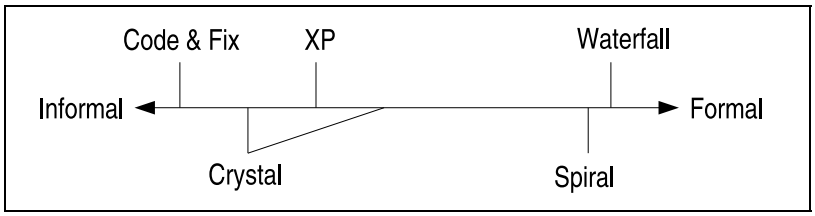


Figure 1: Development methods have different levels of formalism. The Waterfall model (see Section 2.2.1) is very formal, while the Code & Fix method (see Section 2.4.1) is very informal.

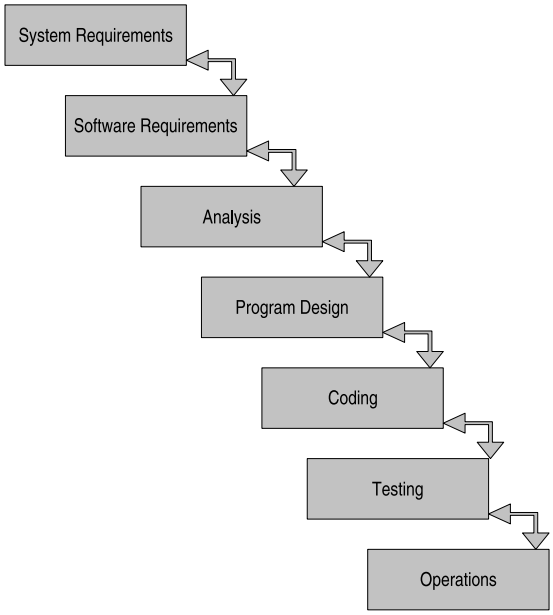


Figure 2: The simplified model presented by Royce [Roy70]. See his article for the final recommended model

methods in general) is based on Boehms [Boe81] Cost of Change Curve 3. The curve shows that the cost of changing requirements or fixing defects rises exponentially as the project nears completion. The argument goes that since changes are much cheaper early in the project, one should use extra time here and with the help of proper planning minimize the neccessary changes later when they are too expensive.

The Waterfall model is a very good example of a plan-driven method is that it clearly states that you should complete program design before the *Analysis* and *Coding* phase. It also specifies that the testing in the *Testing* phase must be planned.

2.3 Iterative methodology

As mentioned above, most plan-driven methods contains some iterations. What separates the iterative methodology from the plan-driven methodology is that while the plan-driven methods create a plan that encompasses the whole project, the iterative methods only plan ahead for the following iteration.

The methods usually have a set of phases that are run through in each iteration. One example of an iterative method would be to take the steps from the Waterfall model and iterate over them until one ends up with a result that the customer is

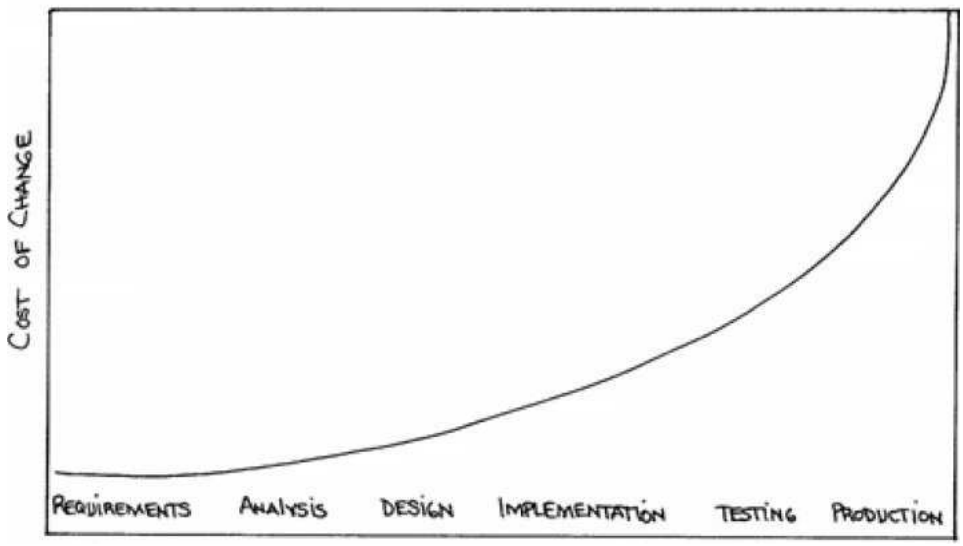


Figure 3: Boehms Cost of Change Curve, a graphical representation of how the cost of fixing defects rises exponentially the later it is done in the development project

happy with.

2.3.1 The Spiral model

Boehms Spiral model [Boe88] is the classical example of an iterative method. It builds on the Waterfal model, but instead of being document oriented it is risk oriented. In each iteration you identify the sub-problem which constitutes the biggest risks, and then you resolve this.

The Spiral model adapts itself based on the risks involved [Boe88]. If for instance a project has a low risk in such areas as getting the wrong user interface or not meeting stringent performance requirements, and if it has a high risk in budget and schedule predictability and control, then these risk considerations drive the spiral model into an equivalence to the plan-driven methodology. However, if a project has a low risk in such areas as losing budget and schedule predictability and control but a high risk in such areas as getting the wrong user interface or user decision support requirements, then the spiral model will adapt and be more equivalent to the evolutionary methodology (see below).

In other words, the various methods discussed in this Section can be coupled with the Spiral model in a natural way [vV00, p62].

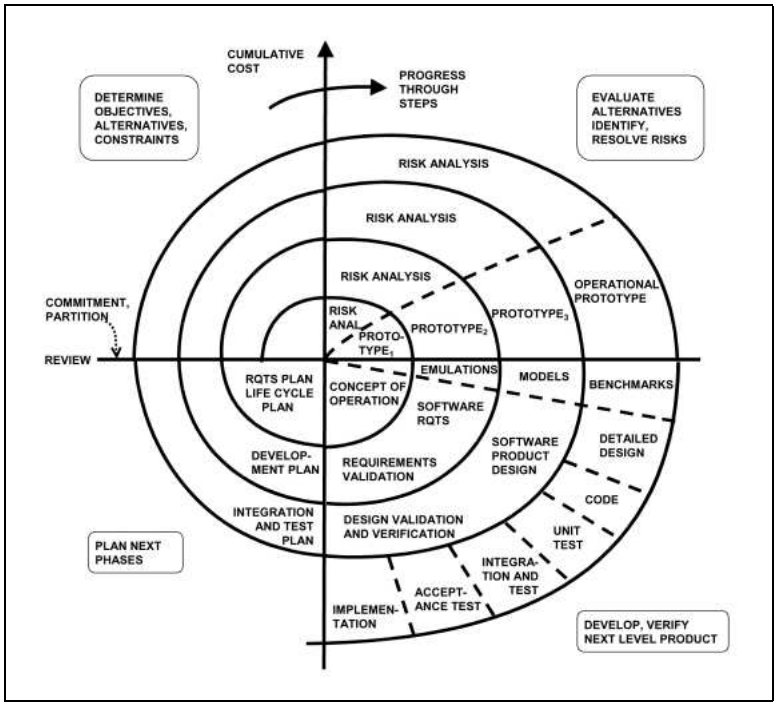


Figure 4: Boehms Spiral Model of the software process

2.4 Evolutionary methodology

The idea of evolving a system is as old as the programming discipline. Instead of using time to think through and plan ahead, the evolutionary methods start the coding phase early and evolves the design as it is needed. The problem with this has often been that the resulting design has been bad or non-existant. The code resulting from such a method has often held low quality and end up being very hard to maintain.

Compared to the plan-driven models, there is no longer any focus on the requirement and design documents. Instead, most of the evolutionary methods focus on letting the development team work close together with the user and discover the requirements based on discussions and cooperation. The user feels more involved in the project and often develops a deeper understanding of what is possible, something that can lead to higher quality requirements.

Hans van Vliet [vV00] writes that if users are shown a working system at an early stage and are given the opportunity to try it out, chances are that problems are detected at an early stage as well. He also writes that if users are in a position to influence and modify the design, the system features will better reflect their requirements and the system will be easier to use.

2.4.1 The Code & Fix method

Maybe the oldest and least formal development method. In short, the programmer sits down and writes code he believes will solve the problem at hand. Then based on feedback he tries to improve (fix) the code. This method is also called hacking, and many professional developers doesn't consider it a proper method at all.

2.4.2 Evolutionary prototyping

McCracken and Jackson [MJ82] gives the following description of what is now usually refered to as evolutionary prototyping

> Some response to the user's earliest and most tentative statement of needs is provided for experimentation and possibly even productive use, extremely early in the development process–perhaps 1% of the way into the eventual total effort. Development proceeds step-by-step with the user, as insight into the user's own environment and needs is accumulated. A series of prototypes, or, what is perhaps the same thing, a series of modifications to the first prototype, evolves gradually into the final product. Formal specifications may never be written. Or, if specifications are needed, perhaps to permit a re-implementation to improve performance, the prototype itself furnishes the specifications.

2.5 Agile development methodology

The dictionary[1] defines agile as follows

> ag·ile (adj.)
>
> 1. Characterized by quickness, lightness, and ease of movement; nimble.
> 2. Mentally quick or alert: an agile mind.

The term agile was chosen in 2001 by the Agile Alliance [All01]. The founders of the Agile Alliance were all working on different software development methods that tries to handle the problem with unpredictable and unstable requirements.

Sometimes refered to as lightweight (in contrast to the plan-driven methods which we earlier termed heavyweight) methods or adaptive methods, the agile methods assumes that there will be unpredictable changes during the development and that it is better to focus on ways to adapt and handle these changes instead of

[1] We used `www.dictionary.com` [Online; accessed 17-December-2005]

trying to predict them. Beck [Bec00] uses the term *embrace change* to describe this attitude.

While the Agile methods are evolutionary, there are several things that bind them together and supports the choice of gathering them under a new methodology called agile. The agile manifesto [All01] by the founders of the Agile Alliance tries to define this:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> **Individuals and interactions** over processes and tools
>
> **Working software** over comprehensive documentation
>
> **Customer collaboration** over contract negotiation
>
> **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

See Cockburn [Coc02, p216–218] for an explanation of the agile manifesto and its implication.

One of the things agile methods have in common is the use of very short iterations. Where the Spiral model is usually used with iterations from 6 months to 2 years [Boe88], most agile methods advocate iterations shorter than 1 month [Coc02]. The reason for this is that shorter iterations cause less damage (cost) if the result after the iteration is defective (misunderstandings or design flaws) and therefore more respondant to change. It also makes it easier to accomodate customer collaboration since the customer can be involved and see the progress at the end of each iteration. (Some agile methods, for instance eXtreme Programming, recommend having a representative from the customer available during the iterations as well.

2.5.1 eXtreme Programming (XP)

XP is a method for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements [Bec00].

XP uses it's so called enabling practices as described by Fowler [Fow01]. These practices are there in an attempt to flatten the change curve shown in Figure 3 enough to make evolutionary design work. One of the goals with these practices is to fail fast, i.e. discover errors/defects as soon as possible. As a consequence, the change needed to correct the defect can be done as low on the cost of change curve as possible.

Later argumentation by Cockburn [Coc00] claims that XP in fact doesn't flatten this curve but that the flattened curve could be seen as a representation of the Cost of Change in XP compared to plan-driven development. However, some of the practices used in XP can be used in plan-driven development as well, and they should give some of the same benefits. Especially pair-programming and test-driven development are easy to transfer.

2.5.2 The Crystal methods

Cockburn [Coc02, p197–212] has created a family of methods with the intention that when you implement one of the methods you adapt it to your situation.

The methods are named after different colors and types of minerals. For instance you have clear quartz crystal which is a light soft method. Then you can have red diamond crystal which is a darker and harder method. The colors corresponds to the size of the project team and the hardness corresponds to the criticality of the project.

Cockburn puts a lot of focus on how you should adapt the method to fit your need. He claims that all methods should be adapted to the situation at hand and the result of failing to do this will effect in higher cost without any increased productivity.

3 Communication

Communication is the key to victory. – *Anonymous*

According to Cockburn [Coc02] software development is a cooperative game of communication, and that this implies that a project's rate of progress is linked to how long it takes information to get from one person's mind to another's. Beck [Bec00] claims that problems with projects can invariably be traced back to somebody not talking to somebody else about something important.

Most of the Agile software methods depend on having the development teams work close together in an open landscape setting, like for example the "caves and common" room layout presented by Cockburn [Coc02]. Beck [Bec00] describes a setting much like the "caves and common" and claims it to be the best. He also mentions that there should be a place for people to relax since taking a breather very often helps when you get stuck while developing.

3.1 Awareness

One of the reasons why Cockburn and Beck recommends this room layout is to improve the awareness within the development team. When someone have short conversations "across" the room, the others in the room will overhear the conversation and subconsciously register what is being said. They will know who asked what and who answered. This is called awareness. With increased awareness, you increase your general knowledge about who knows what. The result is that less time is spent looking for answers since you'll know where to find them.

3.2 Levels of Communication

Cockburn [Coc02] describes different Modalities in Communication. We've devided these modalities into three levels, based on how many of the human senses are used for the communication. We do not consider the sense of smell and the sense of taste to be important in communicating about software development.

3.2.1 Level 3

We start out with the three remaining senses: touch, sight and hearing.

When you talk to somebody in person, you make use of many different mechanisms to communicate. Not only what you say, but how you tilt your head, how you wave your hands and the expression on your face becomes part of what you're communicating. Sight is obviously an important part of the conversation.

Maybe what you're trying to say is hard to express so you find a piece of paper or a whiteboard and make a small sketch while you're talking. The person you're talking to can easily interrupt or ask questions by making a noise, a slight touch or a gesture that indicates that he wants to speak. So touch is used both to interrupt the flow of conversation as well as for drawing or in other ways illustrating something using objects in the room.

3.2.2 Level 2

With distance, the sense of touch is no longer available. Using touch to interrupting the conversation is considered rude or unwelcome by some people, so it is often not used anyway.

As for illustrating by drawing or manipulating objects, the effect can often be duplicated by a video link. With an electronic whiteboard, for instance the MicroTouch Ibid[2] or the Mimio Board[3], you can share a whiteboard across offices.

Still, Cockburn [Coc02] claims that experience shows that communication at Level 3 is better than communication at Level 2. We have not looked at any studies that can explain why this is so.

3.2.3 Level 1

If we remove sight, we only have hearing left. With sight we loose many of the finer details of communication. It is now harder to judge the mood of the person you are talking to, and it's difficult to judge whether calling someone will interrupt their work or not. [Coc02]

3.2.4 Level 0

This is where instant messaging and email comes in. We have no direct connection to the party we're communicating with. All communication is now done via text messages (may contain pictures) or asynchronous audio/video like voice mail or recordings or similar.

[2] `http://www.microtouch.com`
[3] `http://www.virtual-ink.com`

4 Open Source Development

Open Source Software (OSS) is defined by the Open Source Initiative[4] and while the official definition[5] is quite strict, we feel that the following simplification is sufficient for this paper

Open Source means that the source code of the program must be freely available and the license must permit modifications and derived works to be published under the same license

In addition to the Open Source Initiative there is also the Free Software Foundation. While there are differences between these two factions we do not consider them relevant to our paper.

4.1 Structure of Open Source Projects

A lot of the Open Source development efforts can be generalized into three different project structures: Distributed development, partly distributed development and one man efforts.

4.1.1 Distributed Development

Examples of this structure can be found in the Gaim project and the Apache HTTP Server project. Looking at the "Contact Information" page on the Gaim[6] website shows a list of contributors and developers that have no apparent connection except that they are all contributing to the Gaim project. The "About" page on the Apache HTTP Server[7] website says

> [...] The project is jointly managed by a group of volunteers located around the world, using the Internet and the Web to communicate, plan, and develop the server and its related documentation. [...]

The point is that there is no one company in control of the project. It's all distributed, except if two or more of the contributors decide to meet and work together.

[4] `http://www.opensource.org`

[5] `http://www.opensource.org/docs/definition.php` [Online; 19-December-2005]

[6] `http://www.gaim.org`

[7] `http://httpd.apache.org/`

4.1.2 Partly Distributed Development

This structure is typically found in projects that are funded by a company or some other agent because they need the software. In some cases they decides to run the project as an Open Source project.

Examples of this is the SciCraft project[8] run by the Chemometrics and Bioinformatics Group (CBG) in the Department of Chemistry at Norwegian University of Science and Technology (NTNU). This project has a small team working full time on the project. External contributions are welcome, but not neccessary for the project to succeed.

Another example is the NAV project[9] funded by NTNU and Uninett. This project is mainly developed by the centralized networks department at NTNU. It is Open Source and external contributions are welcome but again not neccessary for the project to succeed.

4.1.3 One Man Efforts

Many Open Source projects are results of an idea that a single developer has and decide to act upon. The developer starts a project and after a while he/she feels that this might be of interest to others. Various reasons why the author wants to release the work as Open Source is covered by von Hippel [vHvK03].

Few of these projects amount to anything. Maybe it gains a few users, and maybe some of them sends in a bug report or a patch or a friendly email asking for a feature or change. But in most cases the development is mainly done by the creator of the project.

If the project gains a large user base then this user base usually contains developers that, when encountering errors, can deliver valueable bug reports that the main developer can use to easily fix the problem. When this part of the user base grows big enough we no longer consider the project a One Man Effort, even though there is only one core developer.

4.2 Open Source Development Methodology

In Section 2 we covered most of the predominant development methodologies used for development in a sponsored development project. Now we will look at what simliarities we find between these and what we find in OSS projects. We will only consider the Distributed Development structured projects since the One Man Efforts are to small to be of interest and the Partly Distributed Development efforts can be considered part of the sponsored development projects.

[8] `http://www.scicraft.org`
[9] `http://nav.sf.net`

It is apparent that most OSS projects follow an evolutionary development method. One of the main ethos of the OSS world is (according to Raymond) *release early, release often* [Ray01]. In other words, we can place the oss methodology in under the evolutionary methodology.

What separates the OSS methodology from most other methodologies however is it's distributed nature. Most of the communication happens at Level 0 (see Section 3). The formalism in OSS is low due to it's evolutionary nature, but since the communication happens at Level 0 it has a higher level of formalism than most of the other evolutionary methods. Most of what is being said is said on a mailing list or some other public and static forum. This is maybe one of the positive aspects of the oss methodology. As was discussed in Section 3.1, knowing what the other developers are doing and who knows what is important to achieve effective communication. These communication channels also helps avoid duplicate work since developers usually send emails about what they intend to work on before starting, as mentioned by Østerlie and Rolland [Øs03].

Another thing Østerlie and Rolland mentions (quoting Mockus, Fielding, and Herbsleb 2002) is that in OSS projects, work is not assigned but rather chosen by the developers. Since developers gets to choose their tasks, the tasks is done better than if they were forced upon the developers.

Raymond repeatedly says that the most valueble asset to an OSS project (and it's developers) is the users. He sums this up in the following rule [Ray01]

> If you treat your beta-testers[10] as if they're your most valuable resource, they will respond by becoming your most valuable resource.

In addition to this focus on the users, OSS projects are usually used actively by the developers. This makes many OSS projects very user oriented and user aware. Compared to the other methodologies presented in this paper, only the Agile methodology comes close to this tight interaction with the users.

One very obvious difference between OSS projects and regular projects it that OSS projects are basically free. The work is done by voluenteers and so there is no monitary cost involved in the development. This makes the projects more flexible in that actions that are considered too expensive in regular projects will be implemented much more readily in OSS projects. Examples of this is complete rewrites of the program. Brooks states [FPB78]

> Plan to throw one away; you will, anyhow.

This attitude is also partly advocated by Royce in connection with the Waterfall model [Roy70]

[10]Beta-tester means a user that uses an early version of the program that is known to contain defects. The word also implies that the tester sends reports of problems and ideas for improvements to the developer

> . . . arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations areas are concerned.

We have not seen any studies showing how many OSS projects actually does a complete rewrite of the code base compared to Non-OSS projects.

Raymond writes [Ray01]

> My friend, familiar with both the open-source world and large closed projects, believes that open source has been successful partly because its culture only accepts the most talented 5% or so of the programming population.

The name of his friend isn't mentioned in the article, and the claim is maybe not very well documented. But it is generally accepted knowledge that a skilled developer can be more than 100 times more productive than a less skilled programmer. Now if what Raymond says is true, then this might also be a reason why the OSS community has gained the reputation for delivering quality software.

5 Tools

As presented in Section 4, most Open Source Teams are not co-located. I.e. they have to make do with Level 0 or Level 1 communication (see Section 3).

We have already mentioned the use of email as the main channel of communication. In the following we will take a closer look at the tools that are most used in these projects. We first focus on the purely communicative tools. We then look at the tools used for tracking bugs and features. Finally we will look at a tool that has become very valueable in the user documentation for OSS projects.

5.1 Communication

As mentioned in Section 3, effective communication is one of the key factors for a software development team to succeed. Because of this we start out with a presentation of the most frequently used communication mediums in OSS development.

5.1.1 Mailing Lists

A mailing list is an email address that is read by a mailing list software. The software recieves email messages, and, depending on the contents either acts on them internally (subscriptions, unsubscriptions, etc.) or distributes the contents to the users subscribed to the mailing list. [Wik05a]

Other features that are available in some mailing list software is archives of messages sent to the list and filters that stop unwanted messages, e.g. SPAM.

As described in [CGS04], mailing lists often stand as the primary communication channel for OSS projects. Usually there is several lists where each list serves a specific purpose. For smaller projects, there is normally a list for users and another list for developers. The former list is intended for support and interaction with users while the latter contains discussions about the development and direction of the project. Larger projects tend to have more lists to make it easier for users and developers to get a better signal to noise ratio. An examples of this is the apache httpd project which has 11 mailing lists[11].

Many mailing lists require you to be a subscriber to be able to post to them. Since being subscribed to a mailing list means that you get messages to your email address, it can be argued that mailing lists are too intrusive for casual visitors. The Web Forums described further down might be a reaction to this.

[11] According to `http://httpd.apache.org/lists.html` at the 14th of December 2005

5.1.2 Newsgroups

Despite the name, this is usually a discussion group within the Usenet system. Even though this system is still in use, it's popularity seems to be decreasing in favour of web forums. This might be because newsgroups require the user to use a news application while web forums can be read with the users regular web browser. [Wik05c]

5.1.3 Web Forums

Often refered to as Internet Forums or just forums. This is a facility on the world wide web for holding discussions. [Wik05b]

Compared to Mailing Lists, web forums are often easier to browse and join for casual visitors. The reason why they are easier to browse is that while the mailing list archives are sorted by date, the forums are sorted by topic. The reason why they are easier to join is that the formality of the forum is usually stricter on mailing list [12]. Forums are less strict and can in many cases be considered more friendly to beginners.

Another important difference is that web forums can be considered much less intrusive than mailing lists since the user has to actively visit the forum to read it. However, many developers and very active users consider it bothersome to have to actively visit the forum to find out what is going on.

5.1.4 Internet Telephony

There are many different applications for internet telephony. The most well known is probably Skype [13]. The advantage with Internet Telephony compared to regular telephone services is that in most cases it is completely free.

5.2 Issue Tracking

When a user or a developer finds an error in the program, a so called bug, it is important that the bug is delt with. If a developer finds the bug, and if the developer has the time and knowledge, he might fix the bug at once. Otherwise, if he's busy or he can't fix the bug, he has to pass on the information regarding the bug so that others can deal with it. This is also the case if the bug is identified by a user.

While there exists quite many different issue tracking systems, we have looked at Bugzilla and Trac.

[12] See for instance `http://tgos.org/newbie/rules.html`. Note that most of the usenet etiquette guidelines are also considered to be valid on mailing lists as well.

[13] `http://www.skype.com`

5.2.1 Bugzilla

Bugzilla is one of the oldest OSS issue trackers. It was created by the Mozilla Foundation because the Foundation saw the need for such a system and there were no attractive OSS alternatives available at that time. Bugzilla has grown quite large and is used by a lot of different projects, both OSS and non-OSS.

5.2.2 Trac

According to the official web site, [14] Trac is an enhanced wiki and issue tracking system for software development projects. For information on wikis, see page 19.

In addition to being an issue tracking system, Trac is tightly integrated against the revision control system[15] and a Wiki meant to assist in organizing and documenting the project progress.

5.3 Documentation

One of the things that have been lacking in many OSS projects is user documentation. Writing documentation has been considered a waste of time by some developers and the attitude that if you want to know how to do something and it's not in the documentation, read the source code, has been one of the things that have given OSS projects a reputation for being less than user friendly (see for instance West [Wes01]).

The last few years have put more focus on the writing of user documentation in OSS projects. One of the more popular ways of organizing this documentation i the use of the so called Wiki.

5.3.1 Wiki

A Wiki is (according to Cunningham, the creator of the first wiki [Cun05]) a type of website that allows users to add and edit content and is especially suited for constructive collaborative authoring.

The following storyline explains why the Wiki has become a valueable tool for user documentation in OSS projects.

A user wonders how to use a program he has just installed. He visits the programs corresponding web page and finds a documentation wiki. Here he finds a small tutorial written by an earlier user. The tutorial is written from the eyes of a user who used a lot of time figuring things out on his own and who decided

[14] `http://www.edgewall.com/trac`

[15] The current version only supports the subversion revision control system. For information on revision control systems, see `http://en.wikipedia.org/wiki/Revision_control`

to write this story in the wiki since there was no such story when he tried to find help there earlier. The user who documented his story did so because he likes the program and because he wants to help other users like him avoid spending a lot of time figuring things out. The new user reads the tutorial and finds that it is helpful in getting started. He might correct a few spelling mistakes while he reads it. Later, this user learns of new and interesting features with the program. He remembers how helpful he found the tutorial and decides that maybe it would be helpful if he adds what he knows to the tutorial.

This story is based on my own experience with using OSS tools. Von Hippel and von Krogh gives a good explanation of the motivating factors for joining and supporting OSS projects [vHvK03]. Users who can't help the development can contribute by writing user documentation and providing help to less experienced users. This is made much easier by the Wiki techonology since the effort to join is much lower than with regular documentation technology.

References

[All01] Agile Alliance. Manifesto for agile software development, 2001. [Online; accessed 17-December-2005].

[Bec00] Kent Beck. *eXtreme Programming Explained*. Addison-Wesley, 2000.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Boe88] Barry Boehm. A spiral model of software development and enhancement. In *IEEE Computer*, volume 21, pages 61 – 72. May 1988.

[CGS04] Reagan Penner Carl Gutwin and Kevin Schneider. Group awareness in distributed software development. *CSCW, November 6-10, 2004*, 2004. [Online; accessed 10-November-2005].

[Coc00] Alistair Cockburn. Reexamining the cost of change curve, 2000. [Online; accessed 18-December-2005].

[Coc02] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2002.

[Cun05] Ward Cunningham. Correspondence on the etymology of wiki, 2005.

[Fow01] Martin Fowler. Is design dead? pages 3–17, 2001.

[Fow04] Martin Fowler. Using an agile software process with offshore development, 2004. [Online; accessed 01-September-2005].

[FPB78] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Softw*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[McC04] Steve McConnell. *Code Complete*. Microsoft Press, 2 edition, 2004.

[MJ82] Daniel D. McCracken and Michael A. Jackson. Life cycle concept considered harmful. *SIGSOFT Softw. Eng. Notes*, 7(2):29–32, 1982.

[Ray01] Eric S. Raymond. *The Cathedral & the Bazaar*. O'Reilly, Sebastapol, CA, 2 edition, 2001.

[Roy70] Winston W. Royce. Managing the development of large software systems: Concepts and techniques. In *Technical Papers of Western Electronic Show and Convention (WesCon)*, 1970.

[vHvK03] E. von Hippel and G. von Krogh. Open source software and the "private-collective" innovation model: Issues for organization science. In *Organization Science*, volume 14, pages 209–223. 2003.

[vV00] Hans van Vliet. *Software engineering (2nd ed.): principles and practice*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[Wes01] Casey West. Turning the tides on perl's attitude toward beginners, May 2001.

[Wik05a] Wikipedia. Electronic mailing list — wikipedia, the free encyclopedia, 2005. [Online; accessed 14-December-2005].

[Wik05b] Wikipedia. Internet forum — wikipedia, the free encyclopedia, 2005. [Online; accessed 14-December-2005].

[Wik05c] Wikipedia. Newsgroup — wikipedia, the free encyclopedia, 2005. [Online; accessed 14-December-2005].

[Øs03] Thomas Østerlie. The user-developer convergence: Innovation and software systems development in the apache project. Master's thesis, The Department of Computer and Information Science, Faculty of Informatics, Mathematics, and Electronics, Norwegian University of Science and Technology (NTNU), 2003.