

Open Source Software Development

Theory and Practice

Alex de Landgraaf
alextrime@xs4all.nl
AI01 student number 1256033

Table of Contents

Preface.....	3
The Free Software Revolution.....	4
The rise of Linux.....	6
The Cathedral and the Bazaar.....	7
Criticism on the Bazaar model.....	11
Morphix: Open Source development in practice.....	13
Conclusion.....	15
References.....	16

Preface

This paper has been written for the theoretical part of the Software Engineering course 2003. In it, an overview is given on the how and why on Open Source development, starting with the history of Free Software development, continuing with the changes in the last decade. After that, a few recent projects are placed against the light of the Open Source development model to show the differences between the (ideal) theory and the (common) practice.

The author of this paper has been fascinated by the way Open Source has been gaining ground since the mid-eighties and has been active in Open Source development for a number of years. This paper must not be seen as a complete overview on the multitude of ways on how these projects tend to be constructed. However, this paper should be seen as an overview on the guidelines that are often used and, more importantly, the reasons to follow them. It shows how the Open Source model compares to the more traditional development methods (proprietary and free), why it works better for certain types of projects and what (not) to expect of it.

The Free Software Revolution

In the early '70's a young student, busy on his physics degree at Harvard, found out about the infamous AI lab [6] at MIT. The student, Richard Matthew Stallman, had a knack for coding and after a short while got a part-time job at the AI lab, for coding improvements to their internally-used operating system. For a decade, Stallman's life was good: the AI lab proved to be a thriving community of like-minded hackers, and after his graduation stayed there. By the early '80's a few spin-off companies were founded, one who hired most of the researchers and students then active in the AI lab. Stallman, vowing revenge, joined the other company and coded for 2 years straight, equaling every new feature their rivals made.

During that time Stallman began thinking about something completely different than that people had seen before. This project would be one that still continues to this day: the GNU project [3]. The project's goal is to create a Free UNIX-like operating system, free in that the source code would be available without cost to everyone, and free in the sense that it would be freely distributable to anyone. Officially starting in '84 with the release of a small programmers tool, the fact that the utilities extended and created by Stallman were of high quality and that changes could be easily made ensured that more and more people were using it on a daily basis. Even better, people were sending back fixes and extensions to Stallman, that in turn improved the utility again. This aspect of the Free Software movement is central to its success: If someone finds a bug, people were able to fix it. If someone wanted a new feature, he could implement it and share it with everyone else.

The GNU project started with a few small tools and Stallman's self-made editor: Emacs. After quitting his job at MIT to work on the project full-time, he was able to make a living by selling tapes with GNU Emacs, and later one of the most important parts of the GNU project, his self-made GNU C Compiler. It too started out as a mediocre tool, but quickly became very sophisticated. Today, GCC is one of the most widely used compilers available under UNIX-like systems.

However, this all wouldn't be possible save for one element in the project: a single license that would protect the source code from misuses, which became known as the GNU General Public License, or GPL. This GPL ensured that all software distributed would have to be available freely as stated above and that modifications to the source would be released under the same, GPL, license. In essence, the GPL was what was considered normal in the time of the AI lab-community. By writing these unwritten laws, Stallman created the fundaments for his own community, centered around the GNU project and the GPL license.

One piece of the GNU-puzzle was yet to be completed, the kernel. As the most central piece of an operating system, it might seem strange that Stallman hadn't started on this earlier. Another hacker from the AI-labs, one that pre-dated Stallman's period there, was Andrew Tanenbaum. As a computer science teacher and later professor, he had made a number of pieces for his students to use, one of them being a kernel: Minix. Stallman

and Tanenbaum discussed making Minix GPL'ed, as Tanenbaum already had the source code printed and documented for his students, but the idealistic Stallman and the perfectionistic Tanenbaum couldn't agree on terms, although they did agree on how the kernel ought to be designed, so in the end Stallman began the GNU Hurd project, one to create the last missing piece.

The rise of Linux

However, then came Linus Torvalds. A student at the Helsinki university, he greatly enjoyed the use of Unix with his friends, playing programming-games on the large systems available. Before long, Linus wanted more: only a few people at a time could access a Unix terminal, and the commercial Unix's cost a fortune. Linus bought a new 386-machine, and started playing around with Tanenbaum's Minix, gradually learning more and more about how it fitted together. Linus slowly began using Minix as a platform from which he tried making his own kernel: thus Linux was born:

Hello everybody out there using minix -

*I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).
... I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)*

Linus (torvalds@kruuna.helsinki.fi) [1]

Tanenbaum's Minix was frozen (he didn't want any features in there that would make it more complicated than as the educational tool it was) and Stallman was busy in his Ivory Tower constructing GNU Hurd. Linus on the other hand quickly was tapping in to the fast community of Minix-hackers. People were interested, a community was being formed around the new kernel, other hackers were adding new features and Linus was aiming for a Unix-like system, instead of just copying Minix. Before long Tanenbaum replied to the growing Linux community (as most of the discussions about Linux were still taking place on the, his, Minix's newsgroup), titling his post: 'Linux is obsolete'. They both started a near-religious flame war, but in the end both apologized and went their own ways [1]. Linux was growing quickly, and before long Linus' kernel outdid Minix in both features and the amount of users: A true Open Source project was thriving, using Stallman's GPL license.

The Cathedral and the Bazaar

How are Open Source projects organized? Why are some so effective and are others quickly abandoned? What are the criteria for a successful project?

These questions have been asked by many developers and managers alike. One of them, Eric Raymond, wrote a book on how Open Source projects are structured called 'The Cathedral and the Bazaar' [2], in which he explains why the Bazaar model used by Linus is so much more effective than the more conventional Cathedral model, where a number of programmers program and release their software when it's finished.

Before Linux, most projects that were Open Source had been using the Cathedral method. Stallman's GNU project can be viewed as a huge Cathedral: everyone was working on his own part, and when it's done it was released out into the open. True, lots of changes were made afterward, and the community behind the GNU project has been quite active in updating it's software, however in the end most of the work is being done by only a few people.

The Bazaar method is totally different. Linus released early, and releases often. He delegates as much as he can. Instead of quietly working on making Linux better, a whole range of people with different views and talents are busy on implementing new ideas and fixing bugs. The Bazaar-model makes a project very dynamic, moving at a speed that Cathedral-builders can't match. It seemed messy at the time, but it didn't turn into anarchy, and gradually became a first-class Unix-like kernel, and together with GNU it became a prominent operating system, running a large portion of both servers and (albeit smaller percentage) workstations.

Eric's report is centered around his own Open Source project, in which he determines a number of factors. The following 8 points are what Eric focuses his report on. In my opinion, some should be followed, but others depend highly on the context of the project at hand:

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

Without the itch, there is no motivation for starting the project. Also, if people are to contribute to the project, they need to be scratching their itches too. This happens very frequently in Linux, as certain groups or companies often would like to see their features implemented and are prepared to allocate resources into the project. Many major corporations either have hired developers or contributed otherwise to Linux.

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

Re usability has been an important issue in software development these last

few decades. The rise of Object Oriented programming, the use of dynamic libraries, having large archives on the Internet with code, all are centered around the idea of reusability. Open Source is centered around this principle too, as using code from one project is just a matter of copying it (with copyright notices, naturally), without the hassle of having to get permission from the original author.

3 *“Plan to throw one away; you will, anyhow.” (Fred Brooks, The Mythical Man-Month, Chapter 11)*

Developers learn a lot during a software project. Re-implementing parts or all of the project seems to be pretty useless, but if continuing with the current code base would cause more problems than starting over, the latter should be considered. As Open Source projects are open-ended in nature, most of the time, there are enough opportunities to do this. However, throwing away code, even if not your own, is something developers find pretty useless. Eric has a strong argumentation on why you should start from scratch, but in any large project it's considered a pipe dream. Instead, reuse the bits that work, and extend and embrace it into your own project. Restructuring code can greatly extend the usefulness of it, but you should intimately know how it fits together if you decide to do this.

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

This can be explained in a few ways: As a new developer, motivated to solve his problems, might join a project to solve his own itches. Another way would be for a project leader, when giving an Open Source project the right atmosphere new users and developers are likely to join, and with them new challenges will await the project. Just keep your eyes open, there are numerous problems waiting to be solved.

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

The most important factor in an Open Source project is for the project leader to be motivated. Without motivation, it is better for the project to give it to someone else who is. This way, the project will remain active and the number of 'forks', where other people start their own project with your project's source, will be small. Fork's generally aren't a bad thing, but as re-inventing the wheel isn't efficient trying to keep a project together is a good way, even if sometimes a project leader will have to make compromises.

Contrary to the Cathedral-method for development, the Bazaar model is more prone to project-politics, which can hamper development. It's more chaotic at times, and when a project continues to grow inter-project relationships calls on a need for diplomacy. Often, flamewars erupt about conflicting ideas and opinions, a true leader knows when it's time to take a stroll around the block and relativate the matter at hand. Consensus might not always be possible, sometimes it's best to propose a fork, they can always be merged if one branch is favored over the other.

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

Users are the most important resources available to an Open Source project. With them, new ideas are thought up, new bugs are found and dealt with. Even more importantly, a lot of users are also developers, getting them to help out is a great way to speed up development.

Eric asked Linus about the Linux development model, and he responded: '*I'm basically a very lazy person who likes to get credit for things other people actually do.*' [2]

Although a joke, it has a core of truth in it that is typical for the Bazaar model: You don't have to do all the work. Don't micromanage a large project, just like a manager in a corporation shouldn't look over developers shoulders every minute. If developers are capable, let them be, and if they aren't you can consider mentoring them, which helps both parties.

Just make sure other people are motivated to use your work as a base from which to solve their own ideas, get people to contribute to your project. Also, an Open Source project doesn't have to start from scratch. Start from someone else's code and build from there. Ask to take over an in-active project. Even forking a project is better than starting from scratch. Even if you plan to redo some or all of the code you borrow, having a base from which to build upon speeds up your project, and in turn gets more users and thus more developers.

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

With normal software development, a product is done when it has been released. If the product was successful, a new version might be constructed, but as a customer you only get what is released at that time. Open Source development, using the Bazaar-model, is completely different in that respect: no matter how bad the first release is, if people are still motivated to use and contribute the project will continue. Releasing often keeps the moral high in a community, as contributors see that they are being taken seriously and development is rapidly progressing.

Listening to your customers. Isn't it ironic that the developers of Free Software often gladly help out people with issues with their programs (it helps to make the program better) and that software companies often hype a product, rush it to the stores and wait for the cash to flow, with patches and a helpdesk to ease the complaints of users not getting what they paid for? Do note that this is stated a bit black-and-white and there are enough examples of good proprietary products and unwilling Open Source developers.

In the proprietary world, a user is a consumer, they should be pushed to buy your product. After that, having the user/consumer update their software every

now and then is vital for the cash flow of the company. This is plain economics. In the Open Source world, users are seen as contributors. They help to find bugs, and in some cases are able to offer ways to fix them. It is true that the technical know-how of an average proprietary software user is much less than that of one using a form of Open Source software, but again: Users are one of the most important resources for software projects.

In the last decade or so, Open Source software has been getting the image of 'It grows on trees'. It's free, lets use it. Users are less inclined to help out, as they take it as natural over time. It takes a good leader that is able to motivate new users to actively contribute. Think 'openness', think 'together'. Yes, it's like Sesame Street. Don't go all the way, do keep tabs on the project and don't let your users run away with you, finding a balance between the two extremes should be your goal. Smaller projects tend to have less users, so integrate them more into your project. Don't be scared to alienate some users, don't neglect your own opinion, but don't ignore your users either.

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

'Given enough eyeballs, all bugs are shallow'. Now this is indeed true, and co-exists with that what has been said before. However, Open Source projects aren't the holy grail when it comes to bug-less software. The actual number of people viewing the source code of a random project won't be a more than that of a regular developer team (who should review each others code, but in a normal setting this is only scarcely done). When you look at the most successful and active projects, the balance tips towards the Open Source-side: More users, more bugs and more bug fixes lead to better software. This is the rationale of #7 above, in the end more active users lead to better software. Even then, don't expect people to actually search for bugs in your source, they will only look for them if they are bothered enough to look for a bug. Like rule #1, also your co-developers (and users) need an itch before they will scratch it. Just motivate them to start scratching when they do feel an itch.

Criticism on the Bazaar model

The use of some of the Bazaar-rules in the previous section aren't totally true however. Some contradict each other, for example.

In the beginning of a project, there won't be a horde of users waiting for your software. The tins full of developers won't get opened the instant you announce your project. Starting a project in the Bazaar-model won't work. Even Linus had a reasonable code base, but the most important factor was: his kernel was the only one available. It filled a gap. Now, however, the road is littered with kernels and projects that didn't make it. Don't expect not to have to do anything on your own. Make sure you fill a gap, work hard on your solution, make sure you have a reasonable base from which others can work with and the project will have a chance of success. Don't neglect a good dose of mouth-to-mouth PR.

Also, make sure you have a good design for your project. This way, people will quickly be able to see what you have in mind, and will either help you or go their own way. Make sure your users and developers know what they can expect from your project, keep it simple and clear. It might be less interesting from a developer's point-of-view, but bouncing around doesn't attract a lot of users. Another aspect is having a project leader that knows how to interact with users. It's a very simple way to either rise above otherwise mediocre programming skills, or to turn a perfectly good project into a deserted wasteland. People like enthusiastic or helpful leaders, not whiners or hermits. Running an Open Source project means more than just writing code.

Another form of criticism is the apparent lack of leadership. In conventional software development, development managers do the following for their team. Stated is how the conventional roles of project managers become redundant or are picked up by the community in the alternative Open Source / Bazaar model of development:

- *define goals* and keep everybody pointed in the same direction

In the Open Source development model, and especially in the Bazaar model, it would be counter-productive to try to achieve this. To have motivated users and developers, having people define their own goals is best, with the project leader or leaders as the ones leading the way for what seems right for them. There will be arguments, there will be differences on certain points, but everyone is equal: If you want to develop something else, or fork the project, or stop development, it's your choice, not someone else's choice. The ones defining the goals are often the ones that do the most work. In ordinary proprietary development, programming is left for the employees. Do you think a manager would actually help with development? A good project leader should do his share of work, otherwise he'll become alienated from his development team.

- *monitor* and make sure crucial details don't get skipped

In Open Source development, everyone tries to get his own goals done. If something is missing, users and developers will notice it. There is no one that looks over your shoulder, as there are no features you must implement. Do make sure there is a way to communicate ideas and todo lists, having a central place for holding the project together is essential.

- *motivate* people to do boring but necessary drudge work

Well, if the problem that is being solved is interesting, the developers and testers will motivate themselves. Open Source is not without drudge work, but what drudge work is for one is the solution to a problem for the other. Some outstanding pieces of coding have resulted from years of seemingly useless drudge work. Even worse, the developers probably enjoyed developing it.

- *organize* the deployment of people for best productivity

The ad-hoc development of Open Source projects tends to settle around the problems that need fixing most. People that are able to help out with development will, people that aren't will move on. There is a form of natural selection among developers, the most able developers will stay around because they are motivated and see the development as a challenge and as a solution.

- *marshal resources* needed to sustain the project

Marshalling resources is the defending of resources allocated to a project. Without it, other project groups will run off with the cash flow. In normal proprietary development this is a natural thing for managers to do. In Open Source development, there are only three factors of true importance: Users, co-developers and time. Of course there is competition between rivaling projects, but as it's not a fight for who gets to keep their jobs the rivaling tends to be good-natured.

Without the need for these 5 tasks, managers become more or less obsolete in Open Source development. Some roles are taken over by the project leader or leaders, others become redundant. The main theme for all Open Source developers however will remain the *fun* that the developers have when busy with their hobby. If there are challenges to be taken, if there are problems to be solved, people will be there to take them on, with a grain of salt and a good sense of humor :-)

Morphix: Open Source development in practice

Ordinary papers on Open Source development would have ended here, however as project leader of my own Open Source project, I'm able to give an inside-view on Open Source development.

Morphix [5] is a project that revolves around the idea of having a modular bootable CD or DVD, easy-to-use yet flexible in adaption. With it, users pop in a CD, they boot their computer and within one or two minutes they have their own desktop environment, ready to use for day-to-day work. The project itself is based upon the very popular Knoppix distribution and actively uses the Debian GNU/Linux distribution's software packages. The project is in my opinion quite successful: tens of thousands of modules are downloaded each release, people can easily start sub-projects and build their own modules, and besides quite a few bugs there is a small but active community, working together and helping each other when they can. Originally based around the idea of saving work for people that want their own CD's based on Knoppix, the project has continued to include a number of components that help people with using Linux. Likewise, new problems and ideas continue to rise up, challenges are slowly but surely dealt with and development is satisfactory.

Having not actively promoted the project, it tends to go around by word of mouth. In the beginning, development was difficult but challenging: dissecting the software used in Knoppix and figuring out how it all works together was *fun*. After that, the official project was started with an outline of how it all should work together: a design that still fits perfectly after half a year of development. The design was so accurate, that people began contacting me where they could download this interesting piece of software. Maybe this was a mistake, as I had to turn down quite a few users just because there wasn't anything to use yet!

After one and a half months of developing the first releases were done. My stance on the releases was, and still is: 'It probably won't work, because we miss this-and-this-and-this, but if you find any bugs or have any interesting ideas don't hesitate to contact us!'. I released at regular intervals with short intermediate periods where the focus was on bug fixing. It worked. People were discussing the issues around the modular design, changes were made, bugs were solved. In one sense the project isn't one of the Bazaar model: I have developed and coded most of the work myself. On the other hand I take all of the users seriously and often see their requests granted in new releases, people are active on their own projects, bug fixes are submitted and the small community tries to move things along. Like all Open Source projects, you can't have a pure Bazaar-model project, as they won't work, but using elements in it does tend to make the projects even more 'Open' than that they usually are. You need a useful software base that stands out of the crowd and is useful enough for others to work with.

Something that I have thus neglected was the promoting of the project. Thankfully, this has worked out reasonably well. This could have been a

reason to abandon the project in the early days: the roads are littered with projects that just haven't quite made it. Thus a new rule can be made:

- Make sure you have something that works, and something that is special. Once you do, don't expect to take over the world within a week. And don't give up prematurely.

There are literally thousands of projects that start each week. These aren't the days of C compilers and kernels anymore. Make sure you have something that appeals to both users and developers, make sure you have something that sticks out of the crowd. Before this project, I've co-developed an instant messaging client. After about 8 months of coding, there was no motivation anymore: only a few users ever tried it out, not one developer helped us out. Why? Because there are many instant messaging clients. Ours didn't add a lot, had quite a few bugs and didn't appeal to a lot of users. After some time, if you see that your project doesn't have a future, putting it on ice is the only option left. Yes, it would be nice to hand over your project to someone else that is interested, but you don't have to wait until someone offers to pick it up. We learned a lot from that project, so in that sense the project has been a success: without it, the Morphix project would not have been started. That gives us another rule:

- If at first you don't succeed, go start something else. There is no point in wasting time over a project that isn't showing any progress. Take what you have learned and start or join a different project.

Do make sure that source code and documentation is available online for people interested. Even 'dead' projects can give people good ideas or help them fix their own problems, and thus make the world a more interesting place by making problems easier to fix.

Morphix too hasn't been made from scratch, as stated before. Use the best of the best for basing your own work on, and make it better. That way, people will be interested, ideas and features can be shared with the original project, and progress will be very rapid. However:

- Know what you reuse. Know it's strengths, know it's weaknesses. Know it's rivals and know the alternatives. Even better: Know the source code. It's a luxury-problem, but there are a lot of projects that offer the same solutions to problems. Use the best alternative for building your own solution.

Conclusion

As we have seen, Open Source development has its differences with the conventional methods. There are advantages and disadvantages to both types, and it is clear that Open Source development isn't a silver bullet, it won't fix all development problems, but all in all it is a viable alternative. If Open Source development will work for a certain project depends on quite a few variables, but considering it as an alternative to the usual methods shouldn't have to be seen as something completely alien.

Also, Open Source development doesn't have fixed guidelines. Some that might have worked a few years ago aren't useful, or even counter-productive now. It is a new and old way of development at the same time, it is a very dynamic and fast-changing method and thus the guidelines in this paper should not be seen as anything more than what they are: guidelines.

References

[1] *Rebel Code*, Glyn Moody

[2] *The Cathedral and the Bazaar*, Eric S. Raymond

[3] GNU Project, <http://www.gnu.org>

[4] Electronic Frontier Foundation, <http://www.eff.org>

[5] Morphix Project, <http://www.morphix.org>

[6] MIT laboratory for Computing Science, <http://www.lcs.mit.edu> and the MIT AI Lab, <http://www.ai.mit.edu>