How to Have a Successful Free Software Project

Anthony Senyard and Martin Michlmayr
Department of Computer Science and Software Engineering
The University of Melbourne
ICT Building, 111 Barry St, Parkville
Melbourne, Victoria, Australia
anthls@cs.mu.oz.au, tbm@cyrius.com

Keywords: free software, software lifecycle, development process.

1 Abstract

Some free software projects have been extremely successful. This rise to prominence can be attributed to the high quality and suitability of the software. This quality and suitability is achieved through an elaborate peer-review process performed by a large community of users, who act as co-developers to identify and correct software defects and add features. Although this process is crucial to the success of free software projects, there is more to the free software development than the creation of a 'bazaar'. In this paper we draw on existing free software projects to define a lifecycle model for free software. This paper then explores each phase of the lifecycle model and agrees that, while the bazaar phase attracts the most attention, it is the initial modular design that accommodates diverse interventions. Moreover, it is the period of transition from the initial group to the larger community based development that is crucial in determining whether a free software project will succeed or fail.

2 Introduction

Prominent free software projects such as Linux [34], Apache [1], and FreeBSD [12] have been extremely successful. Nevertheless, there are many projects which must be considered failures. An obstacle to initiating and collaborating on free software projects is the lack of a formal description of the activities and the different phases of the lifecycle of such projects. The principles of free software development have been circulated anecdotally and are aimed at experienced developers [28, 8]. In this paper a more complete, structured and detailed development lifecycle for free software projects is described. This allows for a more systematic approach to be taken toward the development of free software projects so as to increase the likelihood of success.

In his popular essay *The Cathedral and the Bazaar*, Eric S. Raymond [28] investigates development structures in free software and open source projects in light of the success of the Linux kernel.[1] A metaphor is given which compares traditional software development (ie. where source code is not freely available) to the building of a cathedral and free software developed by a community of volunteers, to a market bazaar. We use the terminology of the cathedral and the bazaar throughout this paper but rather than view these approaches as diametrically opposed we see them as complimentary phases within the same lifecycle. Figure 1 illustrates the three basic stages we believe a successful free software project passes through.

The initial phases of a free software project does not operate in the context of a community of volunteers. Indeed, only successful free software projects make a transition from a traditional, closed project to a community based project. Indeed, it is impossible to originate a project in the bazaar phase. Although the requirement to attract and motivate volunteer developers and to create a community around a project is known, no explanation of how free software projects are actually started has been presented. Yet an analysis of this phase indicates its centrality to the success of the project.

It can be seen that the initial phase of free software projects possesses all the characteristics of cathedral style development in sharp contrast to the later bazaar phase. The initial phase to develop an initial implementation is carried out by an individual or a small team working in isolation from the community [4]. The initial implementation is developed by following the common software engineering activities of requirements gathering, design, implementation and testing without any contribution from the outside. This development process shows tight control and planning from the central project author and has been referred to as 'closed prototyping' by Johnson [16].

However, unlike traditional software projects a free software project has to make a transition from the cathedral phase to the bazaar phase to become a high quality and useful product. This transition is associated with many complications, described in this paper, and which are significant barriers to success. In fact, while there are many examples of successful projects in the bazaar

[1] The terms *free software* and *open source* refer to software distributed under certain licenses [26]. The development model of free software described here refers to a group of volunteers, employing a decentralised organisational structure and communicating via the Internet.

IEEE COMPUTER SOCIETY

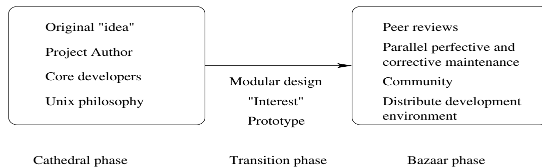

Figure 1. Free Software development lifecycle

phase, the majority of free software projects never leave the cathedral phase and never access the resources of a community of co-developers [6].

The aim of this paper is to provide guidance on how to to start and manage a successful free software project. First, a description of the characteristics of free software projects in the bazaar phase is given to draw out what is currently best practice. This description provides us with the target for the initial phases of free software projects, informs us as to what properties the software should have to facilitate the bazaar phase and outlines issues which must be managed for the project to remain successful. Second, we describe the initial phase (which has much in common with cathedral style development) that must be carried out before a project can reach the bazaar phase. The cathedral phase description will take the form of a linear presentation of the constituent stages as outlined in figure 1. Along with the description, a set of conditions which should be satisfied in order to continue to the next stage within a phase will be presented. Thirdly, we describe the activities which must be performed in the transition phase to allow the project to operate successfully in the bazaar phase. Finally, conclusions on the overall process and lifecycle are presented.

3 The Bazaar Phase

The aim of free software projects is to reach a stage where a community of users can actively contribute to its further development. This section details successful projects so as to define what has worked and what are the pitfalls. From this it is possible to extrapolate what is required in the cathedral phase and the transition phase.

The key characteristic of the bazaar phase is that a community of users and developers are allowed to review and modify the code associated with a software system. The old adage "many hands make light work" is appropriate in describing the reasons for the success of free software. A large number of volunteers working simultaneously on a project has numerous advantages and some problems, which will be outlined in this section. We will refer to the initial implementation as the software which is used at the commencement of the bazaar phase. We will also refer to the transition phase as the steps carried out to put scaffolding in place to support the bazaar phase. These concepts will be expanded upon in section 4.

The bazaar style (illustrated in figure 2) makes source code publicly available and contributions are actively encouraged, particularly from users. Contributions can come in many different forms and at any time. Non-technical users can suggest new requirements, write user documentation and tutorials, or note usability problems; technical users can implement features, fix defects and even extend the design of the software. The key benefit is the software quality which comes from the thorough, parallel inspection of the software carried out by a large community of users and developers.

These benefits are consistent with software engineering principles. The 'debugging process' of a free software project is synonymous with the maintenance phase of a traditional software lifecycle. Maintenance costs of between 50% - 1300% of development costs on a project have been reported [32, 19]. Irrespective of the percentages, it is accepted that software maintenance consumes as much or more resources than all of the previous stages of development. Free software projects, once they reach the maintenance phase and can access a community of co-developers, are much more productive than traditional projects with limited maintenance resources. In this section we will explore what activities facilitate the bazaar phase.

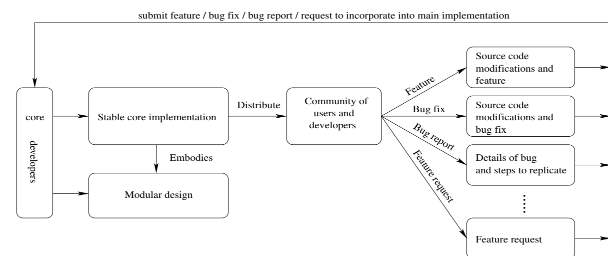

Figure 2. Bazaar style development lifecycle

3.1 Peer Review

Peer review (including inspections and walkthroughs) has been established as part of best development practice and leads to high quality software [10]. Free software projects employ decentralised peer-review by allowing anyone full access to their code. While only a limited number of developers can actually work on the implementation of a program [5], there is no limit to the number of people who can inspect the code and search for defects. It is only possible to fix a bug once the associated conditions have been identified. Successful free software projects benefit from a large base of users and developers who inspect code and identify bugs.

3.2 Concurrent Development

Activities within the bazaar phase can be performed concurrently as there is no distinct separation between activities [16]. There are two main types of development parallelism in the bazaar phase. Firstly, several different types of development can be performed concurrently, such as feature addition and bug fixing. Volunteer developers can perform their favourite activities whenever they wish. This increases the potential usefulness of volunteer developers to the project. Secondly, multiple developers can perform implementation work in parallel. This allows numerous features and fixes to be added quickly. Other tasks, such as documentation and testing also require less coordination and can be performed in parallel [23, 30]. However, there is a limiting factors on

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

the amount of implementation parallelism which can progress the project [5]. This limiting factor is dependent on the modularity of the design of the initial implementation. A second problem is the gradually divergence of developer code from the main code base which requires regular synchronisation.

3.3 Opening Up Requirements

The bazaar phase is characterised by an open process in which input from volunteers defines the direction of the project, including the requirements. The initial implementation is mainly based on the project author's requirements. In the bazaar phase, projects benefit from the involvement of a diverse range of developers (with different requirements) who work together to increase the functionality and appeal of the software. This leads to the problem of feature creep. To prevent this, the maintainer must decide whether a specific feature is in line with the overall scope of the project and the design. If they reject a feature, the volunteer will be discouraged from using and contributing to the project because it does not have a feature they desire. On the other hand, if all features are incorporated the associated feature creep can result in large and potentially unmaintainable software. This problem can be overcome by having a clear and well communicated scope for the project established in the transition phase.

3.4 Parallel Implementation and Debugging

Since many different people can contribute code to a free software project, there should be guidelines which describe the coding style and standards. The GNU Project has an elaborate document describing the best practice and most bigger projects have similar documentation or refer to well-known standards [17]. It is the task of the maintainer to make sure that patches conform to these standards. Patches can come from different people and vary greatly in size. In many projects, there is a small number of frequent contributors who do most of the implementation while a much larger group submit smaller patches to fix defects.

Debugging requires a certain level of technical expertise from the user. In the past, those using free software were power users and typically programmers themselves and this contributed to the practice where users took part in the software inspection process. This has changed as free software has become more popular in recent years. While there are still many technically oriented users, there is an increasing number of users who do not posses the technical skills to inspect the code or fix defects but who can still submit traditional bug reports. The advantage of free software over proprietary software is that anyone can inspect the code and complete a bug report. There is no dependency on the maintainer of the software to fix the bug. Rather, there can be a large number of developers whose task it is to act as a layer between the maintainer and the user. Increasingly, this task is performed by the vendor who puts together complete software distributions based on free software.

3.5 Design and the Importance of Modularity

As additions to the requirements and code base are made, the design must be adapted. Volunteer developers provide a sounding board for doing this. Design alternatives can be discussed and tested in parallel as developers, motivated by the desire to create the most elegant designs and implementations [27], compete to produce the best designs.

A pitfall of adapting the design is that the complexity of the design will increase. In evolving programs complexity increases unless there is specific work performed in order to reduce it [18]. The only way to facilitate feature addition and avoid increasing design complexity is with a modular design from the initial implementation [2, 33]. A clear and modular design of the initial implementation allows the community of developers to refine the requirements and design within a specific scope and is crucial to the success of the project [34].

However, at some point it may be technically desirable to replace a system with a simplified version rather than to continue with the existing system. The need for a complete redesign cannot be addressed through any properties of the initial implementation. There are several factors which promote system redesign. First, the expanding scope of the requirements (from those initially gathered by the project author) and parallel development combine to make the initial design obsolete. In many cases, insights into what the design should have been are gained during the bazaar phase. Furthermore, the revised requirements can be so different from the original ones that the design is simply not modular and extensible enough to easily incorporate the new features. This was the case with Apache which was based on the NCSA http daemon and hence on a design more than 10 years old. Second, there is a great incentive in the free software community to achieve technical excellence [27]. This desire means that if a complete rewrite fulfils this goal, then the effort will be made. This was one of the motivations when Perl 5 was written [16]. Third, there are no constraining economic requirements and so the desire for technical excellence can be satisfied. In free software projects, a complete redesign can be carried out even if iterative development is still possible and easier in the short term.

In most cases, a complete redesign and reimplementation will be performed in a manner similar to the creation of the initial implementation development (the cathedral phase). A core team (consisting of the best developers found in the bazaar phase) will work together to create a totally new design and implementation. This new initial implementation must be more modular and adaptable than the previous software. In summary, the key property of the initial implementation from the cathedral phase (whether the initial or the redesigned software) is a modular and extendible design.

4 The Cathedral Phase

Traditional software development is usually performed by experts working in isolation from their users. In this approach versions of the software are released after the development lifecycle (illustrated in figure 3) leaves the test phase. This approach to developing software is marked by a static development organisation and is accompanied by central planning efforts. Development is driven by a team of individuals and users do not contribute nor access source code. The main characteristic of the traditional lifecycle is the release of new software versions at the completion of

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

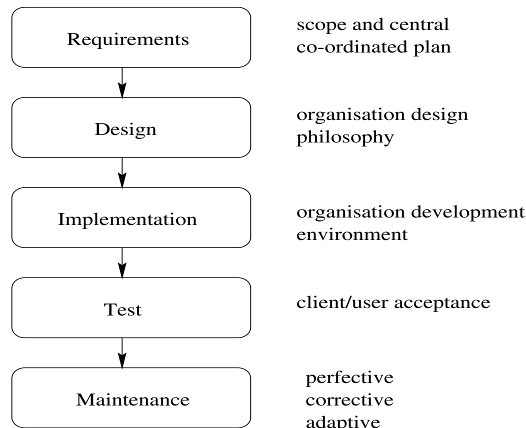

Figure 3. Traditional style development

testing. It is this approach which best characterises how the initial implementation is developed.

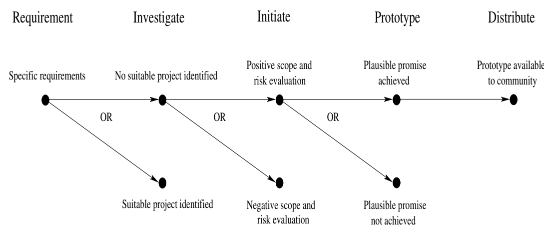

Figure 4. Detailed Cathedral phase

4.1 Requirement

The only condition to be satisfied to move to the investigative phase is that a user has some requirement which could be fulfilled by software. Starting a new free software project is a serious undertaking and involves a long-term commitment of time and energy. As such, free software projects can not get started unless there is a strong incentive for doing so. For an individual to begin the process of setting up a new project this motivation is generated by a specific requirement. In Raymond's word, a project gets started in order to "scratch an itch" [28].

4.2 Investigate

There is one basic condition to move to the initiate stage; that no suitable project has been identified which can satisfy the users needs. Alternatively, a suitable project may have been identified but the user does not want to participate in the project.

The investigation stage ensures that no project has already fulfilled the individual's need. Thus, the individual starts an investigation of existing projects in order to find out if someone else has completed or is working on a tool which fulfils their need. The ideal scenario for the individual is to discover a project which does exactly what they want. The project also benefits since it has attracted a new user who will test the software through use, possibly suggest new requirements and publicise the project.

If no such project is found, it is possible that the individual might instead discover a project which does not yet offer the specific feature they are looking for but which has a similar scope and requirements to those of the individual and which offers a 'plausible promise'. In this case, the individual can decide to participate in the project and request the specific feature or modify the software to implement the desired requirement. This scenario explains how existing projects attract new volunteers and profit from the community. The individual also benefits because it is much easier to get involved in and contribute to an existing project than to actually set up your own project. According to an extensive study of free software developers, more than 30% got involved in the free software community in order to improve products of other developers [13].

However, it might also be the case that no project exists with similar requirements or that a similar project exists but the individual deems that it is not worthwhile to participate in this project. Although the Unix philosophy (with which the free software community is often associated) promotes software reuse [29], there are several psychological barriers which prevent this. Firstly, the project might be written in a programming language the individual has not mastered or dislikes. Secondly, a developer might not agree with the design decisions made by a project. Thirdly, they may dislike the individuals involved in the project. These factors can provide sufficient motivation for an individual to start their own project. This attitude can result in duplication of effort and to the existence of two mediocre software projects while one superior software project would have been possible had the two efforts joined forces. In some situations starting a new project can be beneficial because of a fundamental flaw with the existing software.

4.3 Initiate

There are two conditions which need to be satisfied to move to the prototype stage. The first is that some analysis of requirements, risks and schedule has been performed, consciously or unconsciously. The second is that the would-be developer is sufficiently motivated by the implementation task to proceed.

Individuals who wish to start their own project should undertake (consciously or unconsciously) risk analysis regarding the potential for a successful project, requirements scoping and schedule creation. Firstly, the risk analysis should address the individual's motivation and commitment to a long-term endeavour. Secondly, the risk analysis should address whether the new project can compete with existing ones and attract volunteers. The requirements scoping should define the extent of the project and identify how the new project can differentiate itself from existing projects. It is also beneficial if the project author has a rough idea of the development schedule. This allows other core developers (if they exist) to help with the initial implementation.

This work is done informally as there are no official guidelines for risk analysis, scoping and schedule creation in free software projects. It is only necessary for the individual to have some positive indications from the analysis, scoping and schedule. However, the more positive these analyses are the more more likely it is that

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

the project will be sustainable in the long-term and attract volunteers to move to the bazaar phase. The initiation stage also tests the motivation of the would-be developer. Those who proceed are motivated not just by the requirements and the potential for a successful project but by the programming itself [15]. These factors together with the initial individual's requirement create sufficient motivation to start the project.

4.4 Prototype

The condition the prototype must satisfy to begin the transition phase is that it implement the project author requirements and is extendible to allow multiple developers to work on the project simultaneously [2, 33].

Once the individual has decided that it is worthwhile to continue it is relatively easy to set up the project. No special equipment is necessary as standard PCs are sufficient as development platforms (in most cases). The software necessary to develop and test software, such as compilers, debuggers and programming environments, were some of the first pieces of free software to be created and are widely available with minimal distribution costs.

Once the project infrastructure has been established, the traditional stages of requirements gathering, design, implementation and testing are followed and the individual moves to the developer role. At this point, there is no rigid structure to the stages and activities of development. Some developers follow the stages linearly while others prefer a more circular prototyping style in which they repeat all or some stages several times. Choosing a development model at this point is the responsibility of the developer(s) [2].

Only when the prototype is finished and the developer(s) decide to move to bazaar style development does it become important to get community feedback and involvement. The properties of the design and implementation of the prototype must motivate others to participate in the project. This is facilitated by a simple, clear and modular design.

4.4.1 Requirements

The condition to move to design is simply that the developer has some an an idea of the requirements of the software to be designed. The requirements for the prototype stem from the developer's specific needs [21, 33, 35]. These requirements and the desire to program provide the intrinsic motivation to move forward. The developer has a good understanding of what they want which is refined during the investigation and implementation stage. Sometimes the aim of a project is to reimplement existing software in which case requirements are derived from that source [21]. Additional requirements come from other users requirements found during the investigation stage and experiences with other software [21].

4.4.2 Design

The condition of the design to proceed to implementation is that it can be extended – this is achieved through modularity and simplicity. The process used within the design stage is not important as long as the design has a number of key properties. Combined with the practices of the individual, the Unix philosophy and culture provides a framework for design which has contributed to the success of free software projects. Dividing a software system into subsystems with clear communication and interfaces allows volunteers to contribute to the project without close coordination [23]. Such an approach is strongly advocated as part of best software engineering practice and can be summarised as "a structure is stable if cohesion is strong and coupling is low." [9].

The sharing of source code is an important factor which contributes to the design and implementation methods used in free software projects. Source code is shared across the broader free software community and it is possible to learn by example from reading other's code. Indeed, developers often study other free software projects in detail and participate in them before initiating their own. This progression instills an understanding of how to design and implement complex software. This practice leads to a certain homogeneity and compatibility in design and implementation style across different free software projects and is an effective way of passing on the Unix philosophy [29]. This is less likely to occur between companies who develop proprietary software (if at all) as source code is seen more as a trade secret and competitive advantage.

4.4.3 Implementation

The condition for the implementation to move to testing is that it is technically adequate and shows "plausible promise" [28]. There is considerable project infrastructure available for the implementation stage. This is due to the emphasis in the Unix philosophy on software reuse and tool support [29]. As free software is often designed and implemented in a modular way, software is often split into different libraries. These libraries can then be used to implement functionality required in the initial implementation. Additionally, during implementation, standard tools used in free software, such as GNU autoconf and GNU make, facilitate building software [24]. The use of these tools leads to portability, another goal of the Unix philosophy, and ensures that the source code of many free software projects is embedded in common infrastructure. These facilitate working on different projects because there is less learning of toolsets required [14].

4.4.4 Testing

The only condition required for the transition phase to commence is that some plausible promise has been achieved with the initial implementation and this has been reinforced through testing.

Before the implementation is released to the wider community, the developer performs some testing. It is the decision of the developer how elaborate this process is. Usually, specific testing tools or methodologies, such as regression tests, are not involved [3, 20]. Rather, the developer decides whether the program works for them. If they are not satisfied with what they have produced, they can go back to the design or implementation phase. On the other hand, if they are convinced that the program is in a state where it can be shared with other people and attract more developers, they can release the initial implementation and initiate the transition to the bazaar phase.

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

4.5 Distribute

In the cathedral phase, questions about distribution and hosting of the project are less important. While an effective means of distribution is crucial when making the transition to the bazaar phase, the initial implementation is usually developed by a single developer or a small team. Sharing the code by e-mail with other developers or putting it on a web site are sufficient for distribution. It is possible to use public free software project hosting sites, such as SourceForge or Savannah, but this is not necessary since community review plays less of a role.

There are arguments for and against using hosting sites from the outset. On the against side, if the the developer loses interest in the project, hosting sites will be filled with dormant projects, making it difficult for users and developers to find active projects. On the other hand, using a public hosting site means that the code is preserved and someone else may resurrect a dormant project. However, it is increasingly the case that finding active projects on large hosting sites is difficult. The need to have an effective means to distinguish between active and dormant projects is becoming an issue for free software projects.

5 The Transition

The transition requires a drastic restructuring of the project, especially in the way the project is managed. There are many questions which have to be considered in making a successful transition. The first question is how the code should be distributed. This question is also related to issues of project infrastructure such as the selection of a hosting site for the project. Secondly, the question of which license to use influences the likelihood of contributors becoming involved in the development process and thereby forming a bazaar. Thirdly, the management style to be used must be established, even if it is an implicit decision. These questions are shown in figure 5 and will be expanded upon in the subsequent sections.

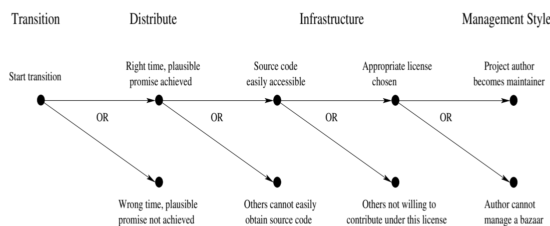

Figure 5. Detailed transition

5.1 Distribution

Commencing the transition at the right time is crucial for a successful project and is a hurdle many projects fail to overcome [11]. Since volunteers have to be attracted during the transition, the prototype needs to be functional but still in need of improvement [16, 28, 2]. If the prototype does not have sufficient functionality or stability, for example if the prototype constantly crashes, potential volunteers may not get involved. On the other hand, if the prototype is too advanced, new volunteers have little incentive to join the project because the code base is complex or the feature they require has already been implemented. For these reasons, the project author must carefully consider when to start the transition and make the code available to others.

5.2 Infrastructure

Assuming that the project author chooses the correct time to start the transition, there are still risks associated with attracting a community for the project which are related to the project infrastructure. The three main pitfalls during the period of the transition concerning the infrastructure are firstly the availability of the source code, secondly the communication mechanisms for the project and thirdly the choice of the license.

Firstly, the infrastructure associated with the project must change to allow for a community to participate in the project and access the source code. The source code must be easily accessible for others so they can download it, compile it, inspect it and ultimately generate patches which can be submitted to the project author. The mechanism to allow this are public hosting sites such as SourceForge or Savannah, or the facilities of the project author. In the past, it was sufficient to have tarballs of the source code but it is increasingly expected that a CVS repository or at least daily snapshots are available [31].

Secondly, project communication mechanisms must be instituted. The need for a community forum to discuss project issues is vital to a successful free software project. The knowledge which is accumulated by the project author during the bazaar phase has to be shared with prospective collaborators. Large projects typically have different forums for user and developer questions as well as a moderated group for announcements. An example of good community forums are public mailing lists, which allow developers to ask questions about the design or implementation and to post patches for other developers to review [14]. To a large degree, public mailing lists and their archives replace formal documentation in free software projects. There is usually no formal requirements specification or design documents for free software projects but the information can often be gathered from mailing lists. Furthermore, volunteers can create frequently asked questions (FAQs) by extracting questions and answers from previous discussions on a project's mailing list.

Thirdly, the source code must be distributed under a license which prospective contributors accept and under which they are willing to share their fixes and enhancements. While this may sound like a non-issue since there are guidelines which say exactly when a license is considered 'Free Software' or 'Open Source' [7, 25], it has been a problem in some projects, including some code related to the Linux kernel. A major difference between the various licenses is how the code can be used in proprietary software. For an overview of different categories of software, such as proprietary and commercial software, see `http://www.gnu.org/philosophy/categories.html`. The appropriate license depends on the nature of the project and requires careful consideration by the project author.

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE
IEEE COMPUTER SOCIETY

5.3 Management Style

Establishing the management style and associated criteria facilitates volunteers joining the project. The most important change to be made during the transition is in the management of the project. In the cathedral phase, the project author or core team makes all decisions on their own. In the bazaar phase, the project author has to open up the development process and allow others to participate in the project. Other developers have to be encouraged and their contributions have to be accepted and rewarded. One form of reward is given when the enhancements or fixes of a contributor are incorporated into the project and a new version is then released. This way, the developers perceive their contributions as useful and will continue to participate in the project. Another important aspect connected with this is to give credit. It is very important to acknowledge contributions properly and visibly, as with a 'THANKS' file.[2]

Unfortunately, changing to a management style which encourages others to get involved can be difficult. Adapting to a new management style and letting others define the direction and take control of 'your' project can be confronting. During the transition from the cathedral to the bazaar style, the whole concept of ownership changes. While the project is clearly controlled by the project author in the cathedral phase, this control must be weakened during the transition and ownership given to the project community. The project author must make the transition from owner in the cathedral phase to head maintainer who incorporates code from others based on certain criteria. Releasing the criteria makes the management transparent and more attractive to volunteers. In fact, there are many different styles as to how projects in the bazaar phase are managed. Two examples, at either end of the management spectrum are described and evaluated through examples taken from free software projects.

Firstly, the rigid management style. The maintenance of the Linux kernel is characterised by relatively rigid control. The reasons for such a management style are connected to the technical risk the project faces. In a kernel, it is vital to avoid feature creep and bloat and to maintain a clear separation between kernel and user space. The main task of Linus Torvalds as benevolent dictator of Linux is to reject rather than to accept code [22]. Centralised control is beneficial in this since it ensures the scope of the project remains focused.

Secondly, the loose management style. In contrast to Linux, KDE is a feature-rich environment where feature improvement and addition are desirable in many different areas. Unlike the case of the Linux kernel, there is only a limited risk of feature creep and bloat. The implication of this is that many people can work on different areas without much interaction or control. Hence, a flat hierarchy combined with self-organisation works well.

There is a diversity of management styles which can be used in the bazaar phase. However, they all share important characteristics. Firstly, contributions from other developers are encouraged and all volunteers are welcome (even for bug identification). Secondly, volunteers can quickly see the impact of their contributions. Thirdly, the project author relinquishes complete ownership of the project and delegates defining the direction of the project to the community. To achieve this an open infrastructure and a license under which volunteers are willing to share their contributions is required.

[2] In a similar way to a 'README' file, it has become a typical convention to have a 'THANKS' file (all uppercase) in the root of a project's tarball. Other common files include COPYING, NEWS and TODO as well as Makefile and ChangeLog.

6 Conclusion

Free software is a phenomenon which has attracted much attention recently. The bazaar phase of such projects can be credited with the feature richness and high quality of free software. The bazaar phase exploits a large number of volunteers who contribute to the development of the software through bug reports, additional requirements, bug fixes and features. However, it has not been previously described how a project can establish this community and be a success. The lifecycle model of free software presented in this paper fills this gap.

We describe a three phase lifecycle for free software projects. The first phase is characterised by closed development performed by a small group or developer with much in common with traditional software development from which we have named the cathedral phase in reference to Eric Raymond. The second phase is a move from traditional development to community based development which we have named the transition phase. Only projects with certain properties can successfully pass the transition phase.

In order to operate successfully in the bazaar phase a number of activities must be completed. The first six are crucial, numbers seven to eleven are important and twelve is desirable:

1. A prototype with plausible promise must have been created.
2. The design of the prototype must be modular.
3. The source code of the prototype must be available and workable (ie. compiles and executes).
4. A community of users and developers must be attracted to the project.
5. The project author must be motivated to manage the project or find a replacement.
6. Project communication and contribution mechanisms must be in place.
7. The scope of the project must be well defined.
8. A coding standard or style must be established.
9. Development versions of software must have short release cycles while user versions must be stable and consistent.
10. A license must be chosen which is attractive to developers.
11. A suitable management style must be selected.
12. An appropriate amount of project documentation must exist.

The final phase is where the project becomes a community based project and gains the associated advantages — we have named this the bazaar phase. The lifecycle model proposed here gives a better understanding of the dynamics of free software and can assist in their success.

A special thanks goes to Damien Wilmann and Dr. June Senyard for reviewing this paper.

IEEE COMPUTER SOCIETY

References

[1] R. Fielding A. Mockus and J. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 263–272, Limerick, Ireland, June 2000. ACM Press.

[2] B. Arief, C. Gacek, and T. Lawrie. Software architectures and open source software – where can research leverage the most? In *1st Workshop on Open Source Software Engineering*. ICSE, 2001.

[3] U. Asklund and L. Bendix. Configuration management for open source software. In *1st Workshop on Open Source Software Engineering*. ICSE, 2001.

[4] M. Bergquist and J. Ljungberg. The power of gifts: Organising social relationships in open source communities. *Information Systems Journal*, 11(4):305–320, 2001.

[5] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, 2nd edition, 2000.

[6] A. Capiluppi, P. Lago, and M. Morisio. Evidences in the evolution of OS projects through changelog analyses. In *3rd Workshop on Open Source Software Engineering*. ICSE, 2003.

[7] Debian Free Software Guidelines. `http://www.debian.org/social_contract`.

[8] C. DiBona, S. Ockman, and M. Stone, editors. *Open Sources: Voices from the Open Source Revolution*. O'Reilly, Sebastapol, CA, 1999.

[9] A. Endres and D. Rombach. *A Handbook of Software and Systems Engineering*. Pearson Addison Wesley, Harlow, England, 2003.

[10] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 1976.

[11] K. F. Fogel. *Open Source Development with CVS*. The Coriolis Group, 1st edition, 1999.

[12] The FreeBSD project. `http://www.freebsd.org/`.

[13] R. Ghosh. Clustering and dependencies in free/open source software development: Methodology and tools. *First Monday*, 8(4), April 2003.

[14] T. J. Halloran and W. L. Scherlis. High quality and open source software practices. In *2nd Workshop on Open Source Software Engineering*. ICSE, 2002.

[15] P. Himanen. *The Hacker Ethic and the Spirit of the Information Age*. Secker & Warburg, London, 2001.

[16] K. Johnson. A descriptive process model for open-source software development. Master's thesis, Department of Computer Science, University of Calgary, 2001. `http://sern.ucalgary.ca/students/theses/KimJohnson/thesis.htm`.

[17] N. Jørgensen. Putting it all in the trunk: Incremental software engineering in the FreeBSD Open Source project. *Information Systems Journal*, 11(4):321–336, 2001.

[18] M. M. Lehman. Programs, life cycles and the laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[19] C. Letondal and U. Zdun. Anticipating scientific software evolution as a combined technological and design approach. In *Proceedings of USE2003*, 2003.

[20] B. Massey. Why OSS folks think SE folks are clue-impaired. In *3rd Workshop on Open Source Software Engineering*. ICSE, 2003.

[21] Bart Massey. Where do open source requirements come from (and what shold we do about it)? In *2nd Workshop on Open Source Software Engineering*. ICSE, 2002.

[22] R. McMillan. Kernel driver. *Linux Magazine*, September 1999.

[23] M. Michlmayr and B. M. Hill. Quality and the reliance on individuals in free software projects. In *3rd Workshop on Open Source Software Engineering*. ICSE, 2003.

[24] A. Oram and M. Loukides. *Programming With GNU Software*. O'Reilly, Sebastopol, CA, 1997.

[25] The Open Source Definition. `http://www.opensource.org/docs/definition.php`.

[26] M. O'Sullivan. Making copyright ambidextrous: An expose of copyleft. *The Journal of Information, Law and Technology*, 3, 2002.

[27] C. Payne. On the security of Open Source software. *Information Systems Journal*, 12(1):61–78, 2002.

[28] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, Sebastopol, CA, 1999.

[29] E. S. Raymond. *The Art Of Unix Programming*. Addison-Wesley, 2003.

[30] D. C. Schmidt and A. Porter. Leveraging open-source communities to improve the quality & performance of open-source software. In *1st Workshop on Open Source Software Engineering*. ICSE, 2001.

[31] M. Shaikh and T. Cornford. Version management tools: CVS to BK in the Linux kernel. In *3rd Workshop on Open Source Software Engineering*. ICSE, 2003.

[32] I. Sommerville. *Software Engineering*. Addison-Wesley, third edition, 1989.

[33] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in Open-Source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[34] L. Torvalds. The linux edge. In C. DiBona, S. Ockman, and M. Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly, Sebastapol, CA, 1999.

[35] P. Vixie. Software engineering. In C. DiBona, S. Ockman, and M. Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly, Sebastapol, CA, 1999.

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY