International Journal of Computing Science and Information Technology, **2014, Vol.2(3), 06-11**
ISSN: 2278-9669, July 2014 (http://ijcsit.org)

A comparative study on Time Based and Feature Driven Release Management Techniques in Free and Open Source Software Projects

Nagendra R [1], Yethiraj N G [2], Manjunatha S[3]
[12]Assistant Professor
Dept of Computer Science, Maharani's Science College for Women
Bangalore, India
[3] Assistant Professor
Dept of Computer Science, Sea College
Bangalore,India
[1]profnagendra@yahoo.com, [2] rajsbmjce@gmail.com, [3] manjus.reddy@gmail.com

Abstract

This paper is a comparative study of release management strategy, time based releases. In contrast to traditional development, which is feature-driven, time based releases use time rather than features as the criterion for the creation of a new release. Releases are made after a specific interval, and new features that have been completed and sufficiently tested since the last release are included in the new version. This paper explores why, and under which circumstances, the time based release strategy is a viable alternative to feature-driven development and discusses factors that influence a successful implementation of this release strategy. It is argued that this release strategy acts as a coordination mechanism in large volunteer projects that are geographically dispersed. The time based release strategy allows a more controlled development and release process in projects, which have little control of their contributors and therefore contributes to the quality of the output.
.

Keywords: Release Management, Time Based Release, Feature Driven

1. Introduction

Free and open source software has had a major impact on the computer industry since the late 1990s and has changed the way software is perceived, developed and deployed in many areas [1]. Free and open source software, or FOSS, is typically developed in a collaborative fashion and the majority of contributors are volunteers. Even though this collaborative form of development has produced a significant body of software, the development process is often described as unstructured and unorganized[2]. This Paper goes through the phenomenon from a quality perspective and investigates where improvements to the development process are possible [3]. In particular, the focus is on release management since this is concerned with the delivery of a high quality product to end-users [4].

2. Problem Description

As discussed earlier, the process of preparing a new stable release for end users is quite elaborate and complex since the software needs to be sufficiently tested, documented and packaged for release. The release management process often faces certain problems, the most common of which are as follows:

• Major features not ready: planning in volunteer teams is very hard and it happens regularly that major features which are identified as critical activities are not ready. These blockers need to be resolved so the release process can continue.

• Interdependencies: with the growing complexity of software, there are increasing levels of interdependencies between software. For example, a piece of software may use libraries developed by another project or incorporate certain software components created elsewhere. This creates a dependency and can lead to problems with a project's release when those other components are not ready.

• Supporting old releases: there is generally a lack of resources in the FOSS community and in many projects old releases receive little support. Some vendors who distribute a given release may step in and offer basic support but it may still be problematic for other users of the software.

• Little interest in doing user releases: even though this study has emphasized the importance of user releases, many developers show little interest in making releases. Since developers by definition generally use the development version they often do not understand the need for a user release or do not see when a user release is massively out of date.Furthermore, some developers claim that preparing user releases only distracts them from the main task they are interested in the development of new functionality.

• Vendors shipping development releases: when projects do not publish new releases, some vendors start shipping development releases because they contain features or fixes which their customers need. This situation is problematic because it can lead to fragmentation and because development releases are generally not as well tested as user releases.

• Long periods without testing: In large projects, there are often long periods in which a large number of changes are made with relatively little testing. At the time of the release, many issues are discovered and this leads to major delays. In the meantime, the last stable release becomes out of date and because of the delay developers may try to push new features into the system, untested.

• Problem of coordination: development in FOSS projects is done in a massively parallel way in which individual developers independently work on features they are interested in. Towards the release, all of this development needs to be aligned and these parallel streams have to stabilize at the same time. Given the size and complexity of some projects, significant coordination efforts are needed. This coordination is difficult, not only because of the size of a project but because participants are volunteers who are geographically dispersed. As in the previous problem, delays can occur and because developers see that there is more time they bring in more changes, leading to further delays.

3. Literature Review

The release management is an important aspect of the development phase and that the general term release management is used to refer to three different types of releases. These types differ quite significantly regarding the audience they address and the effort required to deliver the release.

The three types, which have been identified, are:

• Development releases aimed at developers interested in working on the Project or experienced users who need cutting edge technology.

• Major user releases based on a stabilized development tree. These releases deliver significant new features and functionality as well as bug fixes to end-users and are generally well tested.

• Minor releases as updates or service packs to existing user releases, for example to address security issues or critical defects.

Since developers are experts, development releases do not have to be polished and are therefore relatively easy to prepare. Minor updates to stable releases also require little work since they usually only consist of one or two fixes for security or critical bugs. On the other hand, a new major user release requires significant effort: the software needs to be thoroughly tested, various quality assurance tasks have to be performed, documentation has to be prepared and the software needs to be packaged up. In terms of release authority, it can be observed that major new user releases are typically performed by the project leader or a dedicated release manager whereas development and minor user releases can also be prepared by a core member of the development team. This again shows the significance that user releases have.The interviews also revealed an interesting trend regarding development releases. Traditionally, development releases have played a big role in FOSS projects and it is known that during early development some projects made new development releases available almost every day . Nowadays, actual development releases appear to be of less importance. This change has been prompted by a major improvement and hence deployments of version control systems in recent years. Version control systems are an important development tool in which every change to the source code is recorded. They allow multiple developers to work on the same code base without causing conflicts and provide a historical record of all changes that have been made. Instead of downloading a development release that may be several days or weeks old, developers can obtain the latest code directly from the version control system. The advantage is that it allows developers to track development in real time. A disadvantage is that it is possible that the latest code in the version control system may not work correctly but most projects have policies in place to ensure that such a situation does not persist for a longer duration of time.

Even though version control systems have displaced development releases as the main exchange of source code during daily development, development releases are by no means obsolete. Several interviewees have stressed that while version control systems are a great mechanism to stay up to date regarding the latest code, actual development releases play an important role as a common synchronization point, for example to find out when a defect has been introduced. While development releases and minor updates to existing stable releases play an important function in every project, the main challenge is associated with the preparation of major new user releases.

4. Proposed Technique

The exploratory study presented in has shown significant interest in the time based release strategy within the FOSS community. Even though the projects still face certain problems, the move to a time based strategy can be considered an improvement over their previous release mechanism. In the following, conditions will be discussed that have to be met so that a project can consider implementing a time based release strategy. This is followed by the presentation of an argument based on theoretical propositions as to why the move to this release strategy has led to improvements to the projects under investigation and what are the advantages of this release strategy in general.

5. Evaluation

The following four conditions have been identified that appear to be necessary so that a project can implement a time based release strategy:

1. Enough work gets done: With time based release management, new releases are made according to a certain interval. This strategy cannot be used in projects where there is no guarantee that enough work is done during an interval to warrant a new release.

2. Distribution is cheap: Since time based release management relies on thepublication of regular releases, the distribution of new releases must be relatively inexpensive and easy.

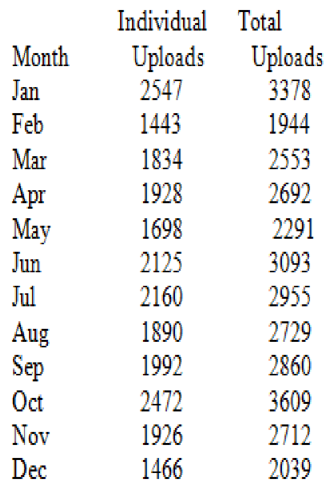
3. The next release does not require specific new functionality: The main focus of time based release management is time rather than specific functionality.It is therefore not suitable for projects in which specific functionality needs to be delivered with the next release, which tends to be the case in traditional software development.

4. The code base and project is modular: It is important for a project to be modular because this allows individual components to be developed, fixed, released, and also taken out again independently.

6. Evaluation Results

1) Enough Works Gets Done

It is important to emphasize the first condition because it rules out time based
releases as a good strategy for the majority of FOSS projects. It has been shown in an empirical analysis of 100 mature projects found on the popular FOSS hosting site Source Forge that the majority of projects are very small (Krishnamurthy 2002). The study showed that only 29% of projects had more than five developers and that, on the other hand, 22% of projects had only
one single developer. With only one or a few volunteer developers working on a project, there is no guarantee that enough changes will accumulate during
a release interval to warrant the publication of a new release. If no changes are made in several months, as is the case in many projects (Howison and Crowston 2004), there is no reason to make a new release. On the other hand, in large projects with hundreds or thousands of developers — which are the focus of this paper — there is always a steady flow of changes (Crowston, Howison, and Annabi 2006). This can be seen in the examples from Debian in figures 6.1 and 6.2. Time based releases are therefore particularly appropriate for large projects.



	Individual	Total
Month	Uploads	Uploads
Jan	2547	3378
Feb	1443	1944
Mar	1834	2553
Apr	1928	2692
May	1698	2291
Jun	2125	3093
Jul	2160	2955
Aug	1890	2729
Sep	1992	2860
Oct	2472	3609
Nov	1926	2712
Dec	1466	2039

the table on the left shows how many individual software packages were uploaded each month and how many uploads were performed in total. The figure above illustrates the total number of uploads.

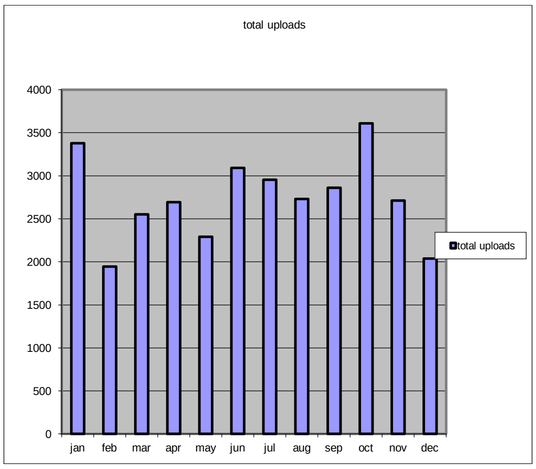


Fig 6.1: Uploads to the Debian archive per month in 2006:

a) Distribution is Cheap

Since time based release management relies on the publication and delivery of regular releases, distribution of the released software must be inexpensive and easy. Since FOSS projects operate over the Internet, distribution of such software meets these criteria. A project can simply put new releases on its web site and vendors will incorporate the new releases into their products and deliver them to customers. While FOSS vendors sell CDs, the majority of software updates are nowadays delivered over the Internet too. For example, the dominant Linux vendor, Red Hat, has established a platform known as the Red Hat Network.2 This web site acts as their 'enterprise systems management' platform through which software updates can easily be installed on thousands of machines in an organization.

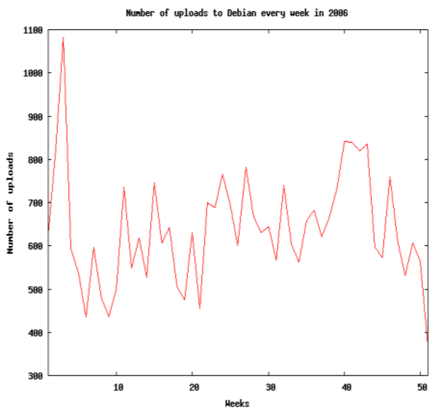


Fig 6.2: Uploads to the Debian archive per week for the year 2006.

b) Specific Functionality is Not Necessary

In contrast to delivery of software over the Internet, shrink-wrapped products sold in shops are associated with higher distribution costs. There are two competing approaches to software production today: some view software as a product industry and sell shrink-wrapped boxes with their software in shops. On the other hand, there is a trend to viewing software production as a service industry. According to this view, revenue is not generated by selling shrink-wrapped boxes but by providing services, such as updates to software:
There's a move towards services and for that a shorter cycle makes more sense. As long as you have a shrink-wrapped product, having a longer release cycle makes sense because you get bigger PR and marketing and people purchase or obtain it. If you have services, you want people to subscribe and then users will get the latest updates and fixes. (Louis Suarez-Potts, OpenOffice.org)

These economic aspects of the software industry are related to the first and third conditions. If a company wants to sell shrink-wrapped boxes, there must be a large incentive for users to buy the new version. Hence, shrink-wrapped software is usually associated with long development cycles and 'big bang' releases, which deliver major new functionality. In order to sell shrink-wrapped software, companies need to add certain new functionality in order to make it attractive to buy the new version. On the other hand, the service industry puts more emphasis on continuous improvements and updates, even if they do not contain major new functionality. For this reason, FOSS projects, which often do not have a commercial interest or follow a service model, need less justification in order to make a new release.

If you work behind a closed door and want to make a big splash when everything is released, it really matters that certain features are in. But for an open source project it doesn't matter if they have features A, B and C. If they have A and B after six months, why would they need to wait for C? If the previous release was fine and they kept everything working and added more fixes and features, why would they not put that out? (Havoc Pennington, GNOME)

The implication of this is that large FOSS projects meet the first three conditions for time based release management: some changes always accumulate in a release interval given that there are hundreds developers or more that and there are fewer considerations to take into account to decide what exactly warrants a new release. This is because periodical updates are viewed positively, even if they only contain small fixes and little new functionality. Finally, distribution of the software can be done over the Internet without much effort or expense.

c) The Code Base and Project is Modular

The fourth and last condition a project needs to fulfill in order to be able to use a time-based strategy is to have a modular structure. This requirement relates both to the code as well as the organization, but they can usually be considered as one factor since there is a strong link between the structure of an organization and the software developed by it (Conway 1968). The reason why a modular structure is so important is that it allows components to be developed, tested, fixed, and released individually. This is important for projects following a time based release management strategy because this strategy requires components to be in a releasable state on a specific date. If there is very strong coupling between a large numbers of components, defects in one component would cause the whole software system to be in a defective state that cannot be released. On the other hand, if the software and its components are highly modular, one can use the current version of most components and either exclude the defective components or use previous versions of them, which do not exhibit these defects.

As a matter of fact, there is strong evidence that large FOSS projects can be considered as an aggregation of many smaller projects. (Crowston and Howison (2005) have found that, As projects grow, they have to become more modular, with different people responsible for different modules. In other words, a large project might be an aggregate of smaller projects.

This insight is very important in terms of time based release management because it allows the implementation of two complementary release mechanisms: individual components are developed independently and can make their own releases as they wish, and the overall release in which all components are combined and tested can be performed with a time based strategy. Such strategies can be observed in a number of projects, for example in Debian and GNOME. In Debian, individual software packages are uploaded on a daily basis. New software is uploaded, bugs are fixed and other modifications are performed. At the time of a major new release of Debian, all software packages which are in a releasable state will be included in the overall release of Debian. Software packages which have major defects will either be fixed or excluded, assuming the software package is not critical or that other packages have no dependencies on it. GNOME, which consists of hundreds of libraries and applications, follows a similar model. In fact, the insight that the project is an aggregate of smaller projects originally led them to the implementation of a time based release strategy after their 2.0 release in 2002.

As GNOME got increasingly bigger it was increasingly like a Linux distribution, meaning it was like a big collection of different software that had lots of different maintainers, different technology and maturity. There was always some stuff that was really stable and some that was unstable. You just had to declare a certain day to release and go with it. (Havoc Pennington, GNOME) The X.org project presented in section 5.7 is another good example of a project exhibiting these characteristics. This project is interesting because it has just recently made the change to a time based release strategy. With their 7.0 release, the project made two fundamental changes. First, they moved from a monolithic code base to a modular one. Second, they decided to move to a time based release strategy. The second change would not have been easily possible without the former. By moving to a modular code base, they allowed individual components to be developed and released independently. Software is developed independently and “once the component is ready, the new version can be included in the next roll-up release” (Kevin E. Martin, X.org) that is done on a six month interval. The modular nature means that the release manager has greater choice as to which components to include in the next release: For the release manager, if you don’t hear from somebody, you can always grab the latest stable version. It requires less coordination because there is now sort of a fall back. If I don’t hear from that project, I will just take what I think is right and ship that even if it’s the same we shipped last time. (Stuart Anderson, X.org) Effectively, this fall back option provided by the modular nature of X.org supports the time based release strategy because it makes it easier to meet the target date.

7. Conclusion

A project employing a time based release strategy with a well planned schedule and regular releases is associated with a number of advantages compared to projects which follow feature-driven development that is often associated with an ad-hoc release style. Four classes of stakeholders are affected by the release process and they gain the following benefits from a time based release strategy with predictable5 and regular releases.

• **Organizations:** the predictability of time based releases allow organizations to plan their software deployment better.

• **Users:** projects which follow a regular release cycle can provide users with fixes and new features periodically.

• **Developers:** contributors know exactly when they have to finish their work in order to make it into the next release. Their contributions will get published and shipped to users relatively quickly, and this may be associated with increased motivation.

• **Vendors:** projects with clear schedules allow vendors that would like to distribute the software to make better decisions as to which releases to use.

References

[1] Krishnamurthy S. (2002). Cave or community?: An empirical examination of 100 mature open source projects.

[2] Howison, J. and K. Crowston (2004). The perils and pitfalls of mining SourceForge. In Proceedings of the International Workshop on Mining Software Repositories (MSR 2004), Edinburgh, UK, pp. 7–11.

[3] Crowston, K., J. Howison, and H. Annabi (2006). Information systems success in free and open source software development: Theory and measures. Software Process: Improvement and Practice 11 (2), 123–148.

[4] Conway, M. E. (1968). How do committees invent? Datamation 14 (4), 28–31.

[5] Crowston, K. and J. Howison (2005). The social structure of free and openSource software development.

[6] Aberdour, M.. Achieving quality in open source software. IEEE Software 24(1), 58-64, 2007.

[7] Amor, J. J., J. M. Gonzalez-Barahona, G. Robles, and I. Herraiz. Measuring libre software using Debian 3.1 (Sarge) as a case study. Upgrade Magazine, 2005.

[8] Erenkrantz, J.R. (2003). Release management within open source projects. In Proceedings of the 3rd Workshop on Open Source Software Engieering, Portland, OR, USA, pp. 51-55. ICSE.

[9] German, D. (2004). The GNOME project: a case study of open source, global software development. Journal of Software Process: Improvement and Practice 8 (4), 201-215.

[10] German, D. M. (2005). Software engineering practices in the gnome project. In J. Feller, B. Fitzgerald, S. A. Hissam, and K.R. Lakhani (Eds.), Perspectives on Free and Open Source Software, pp. 211-225. Cambridge, MA: MIT Press.

[11] Jorgensen, N. (2001). Putting it all in the trunk: Incremental software engineering in the FreeBSD open source project. Information Systems Journal 11 (4), 321-336.

[12] Levin, K.D. and O. Yadid (1990). Optimal release time of improved versions of software packages. Information and Software Technology 32 (1), 65-70

[13] Michlmayr, M. and A. Senyard (2006). A statistical analysis of defects in Debian and strategies for improving quality in free software projects. In Bibliography 208 J. Bitzer and P. J. H. Schrloder (Eds.), The Economics of Open Source Software Development, Amsterdam, The Netherlands, pp. 131-148. Elsevier.

[14] Narduzzo, A. and A. Rossi (2004). The role of modularity in free/open source software development. In S. Koch (Ed.), Free/Open Source Software Development, pp. 84-102. Hershey, PA, USA: Idea Group Publishing.