Time-Based Release Management in Free and Open Source (FOSS) Projects

Martin Michlmayr, Centre for Technology Management, University of Cambridge, Cambridge, UK

Brian Fitzgerald, Lero – Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland

ABSTRACT

As the Free and Open Source (FOSS) concept has matured, its commercial significance has also increased, and issues such as quality and sustainability have moved to the fore. In this study, the authors focus on time-based release management in large volunteer FOSS projects, and reveal how they address quality and sustainability issues. They discuss the differences between release management in the traditional software context and contrast it with FOSS settings. Based on detailed case studies of a number of prominent FOSS projects, they describe the move to time-based release management and identify the factors and criteria necessary for a successful transition. The authors also consider the implications for software development more generally in the current dynamic Internet-enabled environment.

Keywords: *Empirical Research, Open Source Software, Project Management, Software Engineering, Time-Based Release Management*

INTRODUCTION

"Release early and release often."

Despite Raymond's (1999) provocative characterisation of release management in Free and Open Source Software (FOSS), the topic has been the subject of little research in the interim, exceptions being studies by Erenkrantz (2003) and Michlmayr et al. (2007). Furthermore, even in the traditional software literature there have been relatively few studies of release management, most commonly in conference proceedings (e.g. Dayani-Fard et al., 2005; Du & Ruhe, 2005; Erdogmus, 1999; Li et al., 2003; Ruhe & Greer, 2003; Sassenburg & Bergout, 2006) and a smaller number of journal papers on the topic (e.g. Greer & Ruhe, 2004; Levin & Yadid, 1990).

As the FOSS concept has matured, its commercial significance and economic potential has also increased, and issues such as quality and sustainability have become increasingly important (Fitzgerald, 2006). Indeed there is

DOI: 10.4018/jossp.2012010101

evidence that a significant inhibitor to FOSS adoption arises from the perception of a lack of guaranteed quality in FOSS products (Tawileh et al., 2006). As a consequence, FOSS projects need to mitigate risk from such specific issues as lack of deadlines (Garzarelli & Galoppini, 2003), reliance on volunteers (Robbins, 2002; Michlmayr & Hill, 2003) and ad-hoc coordination and management processes (Bergquist & Ljungberg, 2004; Zhao & Erlbaum, 2003). A number of FOSS projects appear to have addressed the above issues through formalising their release management process. The latter is an important part of a project's approach to quality assurance since developers stop adding new features during the preparation for a release and instead focus on the identification and removal of defects. The feedback obtained after a release also provides information as to which parts of the software might need more attention.

Despite the increased company involvement in FOSS, a comprehensive study found that more than two-thirds of FOSS developers comprise individual volunteers (Ghosh, 2006). Consequently, our study focuses on *volunteer* FOSS projects as these will require more formalised processes to mitigate perceptions of risk in adoption. Furthermore, many of the unique and most significant benefits of FOSS arise in *large* projects. For example, large projects are more likely to result in communities forming around them which have sufficient levels of participants with diverse skills to ensure a rapid development trajectory and prompt defect removal (Mockus et al., 2002; Raymond, 1998).

Given the above, our overall research objective was *to investigate release management in large volunteer-oriented FOSS projects*.

The paper is laid out as follows: First, we discuss the topic of release management in traditional software contexts and then discuss its role in the specific context of FOSS projects. Then we present our two-phase research approach for this study. Afterwards we analyses and discusses the findings of the study. Finally, we discuss our conclusions and identify implications for research and practice.

SOFTWARE RELEASE MANAGEMENT

Software maintenance is the sub-field concerned with evolution and maintenance of software after its initial development and release. Levin and Yadid (1990) criticise traditional models of software development which only focus on the initial release and ignore subsequent releases. A continuous release strategy is important for several reasons: it delivers both fixes and new functionality to users (Levin & Yadid, 1990). Also, it staves off obsolescence by ensuring the value of the software is maintained (Baetjer, 1997). As the development environment has become more dynamic and fast-paced, the so-called Internet-time (Baskerville et al., 2002), incremental releases have become more common (Greer & Ruhe, 2004). However, in proprietary release management, this is a complex issue which requires delicate balancing as early introduction of a new release may erode the market-share and revenue generating potential of the existing release.

In FOSS projects such commercially-driven balancing has not been significant to the same extent (Robbins, 2002), and there are other significant differences also. Table 1 provides a summary which contrasts the differences between release management in traditional proprietary development and FOSS projects.

Overall, however, release management has been under-researched in relation to FOSS. While Erenkrantz (2003) has identified characteristics related to release authority and actual work during a release, we know little about actually moving from the development phase to preparation of a release. Fundamentally, it is not obvious how a team of loosely-connected globally distributed volunteers can work

Table 1. Traditional v. FOSS release management practices

Traditional/Closed Source	FOSS
Often follows a waterfall model	Typically follows iterative development practices
Delivery of a monolithic release after long time in development	Small releases published in an incremental fashion
Uses dedicated planning tools, such as Gantt charts	Few dedicated planning tools but good integration of infrastructure (e.g. bug tracking) with release planning
Development releases are private	Development is open, and releases and repositories accessible
Few releases made for the purpose of user testing	Development releases published according to motto "release early, release often"

together to release software, some of which consists of millions line of code written by thousands of people (Gonzalez-Barahona et al., 2001), in a timely fashion and with high quality. There is much evidence to suggest this is a problematic issue. For example, Debian has experienced increasingly delayed and unpredictable releases with up to three years between stable releases. However, this pales into insignificance when compared with cases such as tar and Mutt, an email client, which saw more than five years between stable release, and even the compression utility, gzip, which had 13 years between stable releases (1993–2006). Nor were these products entirely bug-free when the new versions were eventually released.

In recent time, the issue of release management has become an important focus in many FOSS projects, and a number of projects have drastically changed their release strategy, thus prompting our interest in this area.

Coordination Theory

From a theoretical viewpoint, release management highlights the critical importance of coordination. While much FOSS development involves parallel development on self-selected tasks (Mockus et al., 2002), when a release occurs, all development work needs to be aligned and these parallel streams have to stabilise simultaneously. Given the size and complexity of some FOSS projects, significant coordination efforts are needed. This coordination is difficult, not only because of the size of a project but also because the majority of participants are volunteers who are geographically dispersed.

Malone and Crowston (1994) define coordination as "managing dependencies between activities… if there is no interdependence, there is nothing to coordinate". Coordination theory provides an approach to studying processes that are employed by an organisation to perform specific activities and dependencies that occur while these activities are carried out. Processes are coordinated in different ways, but organisations often face similar problems that are managed similarly among different organisations. When actors try to perform their activities, there are dependencies that influence and limit how particular tasks can be carried out. In order to overcome these challenges specific additional activities have to be performed (Crowston, 1997).

When analysing activities, coordination theory therefore makes a distinction between the activities of a process that are needed to achieve the goal and those that are performed to manage dependencies. Coordination theory is mostly concerned with the latter, namely the coordination mechanisms. Some of these coordination mechanisms can be specific to a situation but many are general as they pertain to different situations across organisations.

Malone & Crowston (1993) have proposed a framework which can be used to analyse general coordination mechanisms based on the dependencies they seek to address (see Table 2).

Shared Resources

A resource allocation process is needed when multiple activities require access to the same limited resource (such as time or storage space). An organisation can use different mechanisms to deal with this dependency. A simple strategy would be a 'first come, first served' approach but it is unlikely that this mechanism is appropriate in complex situations because it might stall activities with high importance or urgency. Another way to address this dependency would be to perform bidding within the organisation in a similar fashion to a conventional economic market.

Producer/Consumer Relationships

These dependencies occur when an activity produces something that is used by another activity. This dependency has three different forms:

1. Prerequisite constraints: the producer activity needs to be completed before the consumer activity can begin. A notification process needs to be put in place so the consumer activity can immediately start when the producer activity has been completed. Furthermore, organisations can perform active tracking to make sure prerequisite constraints are fulfilled, for example by identifying activities on the critical path;
2. Transfer: when the producer activity creates something that must be used in the consumer activity, some kind of transfer has to happen. In some cases, the consumer activity is performed immediately after the producer activity is completed, so the output can be used directly without requiring storage. It is more common, however, that finished items need to be stored for some time before they are used by a consumer activity. Hence, an inventory of completed items is often maintained;
3. Usability: what the producer creates must be usable by the consumer. This can be done through standardisation, but it is also possible to simply ask the users what characteristics they want. Another mechanism is participatory design, in which "users of a product actively participate in its design" (Malone & Crowston, 1993).

Simultaneity Constraints

This dependency occurs when activities have to happen at the same time (or cannot occur simultaneously). One mechanism to ensure that dependencies will not interfere is the creation of a schedule. Synchronisation can be used to make sure that events happen at the same time, for example by arranging for several people to attend a meeting at the same time.

Table 2. Activity dependencies and coordination processes (Malone & Crowston, 1993)

Dependencies	Coordination Process
Shared resources	First-come, first-served; priority order, budgets, managerial decision, market-like bidding
Producer-consumer relationships • Prerequisite constraints • Transfer • Usability	Notification, sequencing, tracking Inventory management Standardisation, ask users, participatory design
Simultaneity constraints	Scheduling, synchronisation
Tasks and subtasks	Goal selection, task decomposition

Tasks and Subtasks

This activity can occur when a goal is divided into sub-goals (or activities) which are needed in order to attain the overall goal. It is possible to start with the overall goal and then decompose the task into different sub-goals in a top-down fashion. A bottom-up approach would be goal selection where a number of individuals realise that they could combine their activities to achieve a higher goal.

This framework will be used to integrate coordination issues related to release management in FOSS later.

RESEARCH APPROACH

This study was not concerned with deductively testing some *a priori* defined hypotheses. Rather, the emphasis was on inductively exploring and deriving lessons on FOSS release management from grounded examples of FOSS release management in practice. As McLean (1973) aptly put it: "the proper place to study elephants is the jungle, not the zoo". Research is needed into the actual practice of FOSS release management, justifiable even solely on the basis that practice has often preceded theory in the field. In the early stages of a discipline, theory can best progress by examining good practice (Glass, 1991). Also, given the wide gap between the best and average practice in the field (cf. Boehm, 1981; Brooks, 1987), it is important to discover the essentially good practices of FOSS release management, so that these can be transferred to other projects.

Two-Phase Research Approach

The research was deliberately conducted in two phases. In phase 1, a series of exploratory interviews were carried out with core developers and release managers from 20 diverse FOSS projects. Projects were selected to ensure a wide coverage of different types of FOSS projects (Table 3). Interviews were transcribed, yielding 41,000 words.

Following analysis of this Phase 1 interview data, time-based release management emerged as a significant issue. Unlike traditional release management, which is typically feature-driven, time-based release management makes releases available according to an agreed interval. In phase 2 of the research, seven FOSS projects were selected as case studies to investigate time-based release management in more depth. We wanted to choose "extreme cases" (Miles & Huberman, 1994) which would reveal more detail on the phenomenon of interest. The following selection criteria were used to choose the seven case study projects:

- **Complex.** Phase 1 interviews suggested that coordination is more challenging in large and complex projects. Rather than using lines of code or number of developers, we chose to operationalise this criterion as whether the project had a dedicated release manager or release team;

Table 3. Phase 1 FOSS projects

Projects			
• Alexandria • Apache • Arch • Bazaar • Debian	• GCC • GNOME • Libtool • Lua • Nano	• OpenOffice.org • Postfix • Squid • Synaptic • Template Toolkit	• Twisted • Ubuntu • Vserver • X.org • XFree86

- **Voluntary.** Some definitions of voluntary FOSS projects are based on developers participating in their free time and not profiting economically. However, we operationalised this criterion in terms of control. If the release manager cannot control what someone works on, coordination is clearly more challenging, and we use this to characterise projects as voluntary;
- **Distributed.** Again, geographically distributed projects face more complex challenges in terms of coordination and release management. Thus we chose projects which had a globally distributed developer base;
- **Time-Based.** Given our specific focus on time-based release management, we chose only projects which had already moved, or were currently moving to a time-based release strategy. This also gave us a range of projects at different stages of implementation of a time-based release strategy;
- **Licensed as FOSS.** Our focus is on collaborative development enabled by FOSS and hence we only included projects which had a clear FOSS license, although our projects included those whose origins were in both commercial environment and true community projects.

The selected case study projects are summarised in Table 4 which also shows the release interval. The projects represent a good cross-section of domain and application type.

The unit of analysis here was the individual FOSS project, and at least 3 key experts were selected from each project for in-depth interviews. In this phase, we also sought to interview a vendor representative associated with each project wherever possible. This is important as software vendors rely on FOSS releases to integrate with their systems, and they can thus provide a complementary perspective on release management issues, since they serve effectively as the connection between the developers of the software and the actual users. This research phase resulted in 67,500 words of transcript.

Below we discuss the case study method and the interview data collection approach in more detail. We also discuss the selection criteria for each phase of the research.

The Case Study Method

The case study is not viewed in a similar fashion by all researchers (cf. Smith, 1990). However, according to one of the more common interpretations, it describes a small number of contexts, and usually involves the collection of a large amount of qualitative information. Case studies

Table 4. Phase 2 case study projects

Project	Release Interval	Introduction of Time-Based
Debian	15-18 months	After 3.1 release in June 2005
GCC	6 months	2001
GNOME	6 months	Beginning of 2003
Linux kernel	2 week merge window, releases every 3-4 months	Middle of 2005
OpenOffice.org	3 months	Beginning of 2005
Plone	6 months	Beginning of 2006
X.org	6 months	End of 2006

can be very valuable in generating an understanding of the reality of a particular situation, and can provide a good basis for discussion. There is no attempt at experimental design nor precise control of variables.

In-Depth Personal Interviews

The purpose of the personal interview is to encourage the interviewee to relate experiences and attitudes relevant to the research problem (Walker, 1988). It is a very flexible technique in that the interviewer can probe deeper into any interesting details that emerge during the interview, and concentrate in detail on particular aspects.

A number of problems have been identified in relation to the use of interviewing as a research technique. A frequently-cited problem is that of researcher bias, that is, the researcher may have expectations as to what the research is going to uncover and may ask questions that elicit the answers he or she wants to hear. This may be subconscious, but the way questions are phrased may lead to particular answers being given. The problem of researcher bias is further compounded by another associated problem, demand characteristics. This refers to the phenomenon whereby subjects give answers that they think the researcher wants. Critics of the interviewing technique suggest that the researcher "acts like a sieve which selectively collects and analyses non representative data" (Bogdan & Taylor, 1975). However, almost all research methods are 'guilty' of bias to varying degrees. In the interviews, open-ended questions were used as often as possible to allow more freedom for answers. (The interview guide is available from the authors on request).

Reliability and Validity Issues

Research reliability is concerned with the consistency with which research results can be replicated. A frequent criticism of qualitative research is that due to its subjective nature, replication is problematic. While acknowledging that qualitative analysis would not expect all researchers to interpret the findings in exactly the same way, it is important that the research process be transparent and accessible to others. To help address research reliability, Yin (1994) recommends the use of a case study database and protocol. A case study database was established which contained the raw field notes, transcribed interviews, and coding of this data.

Content analysis was undertaken using grounded theory coding techniques proposed by Corbin & Strauss (1990). This necessitates the researchers to be immersed in the data and to draw on existing theoretical knowledge without imposing a theory (Glaser & Strauss, 1967). It, thus, encourages the researcher to be flexible and creative while imposing systematic coding procedures (Corbin & Strauss, 1990).

The initial stage of open coding involved detailed examination of the field transcripts to ascertain the main ideas. These were then grouped into meaningful headings [informed by constructs developed in the earlier sections] to reveal categories and their properties. Axial coding was then used to determine relationships between categories and their subcategories e.g. conditions, context, action/interaction strategies and consequences. This process continued in an iterative manner, and resulted in the elaboration of several categories and relationships. Analytical memos were written as patterns and themes emerged from these field notes (see the Appendix for an abbreviated example of the above).

The case study protocol specifies the criteria for selecting the case applications, the choice of whom to interview, and the interview protocol in terms of interview questions.

Research validity is concerned with whether the actual research in practice matches what it purports to be about. In interpretive research this is primarily concerned with the "truth value" of the research (Miles & Huberman, 1994).

Construct validity deals with the extent to which the constructs as operationalised relate to the research phenomenon being studied. In

this study, given the lack of research on FOSS release management, construct validity was important. Yin (1994) describes three tactics to deal with construct validity: the use of multiple sources of evidence, the establishment of a chain of evidence, and key informants reviewing draft findings. In this case, the collection of data on the same phenomenon from multiple interviewees over two research phases, both within and external to the projects, together with information gleaned from project documentation and presentations, project websites, and relevant mailing lists, helped address the multiple sources of evidence criterion. In relation to the chain of evidence criterion, this was addressed through the establishment of a case study database, and the rigorous analysis and coding of data. Finally, key informant review and feedback was addressed in a number of ways. A draft of the findings were sent to the key informants interviewed from the projects. Also, the findings were presented at several workshops and conferences attended by several of the project participants and FOSS researchers and practitioners more generally.

External validity is concerned with the extent to which a study's findings can be generalised. One of the limitations of this study might appear to be the fact that it is based on a small number of cases and thus there is limited scope for generalisation. Following this conventional statistical model, researchers have suggested increasing sample size or number of case study organisations, However, Lee and Baskerville (2003) propose four distinct categories of generalising, only one of which corresponds to statistical sampling-based generalisation. One of the other categories in their framework, that of generalising from empirical description to theoretical statements, is more applicable to our research study. This view of generalising from thick description to theoretical concepts, specific implications and rich insight is also recommended as a strategy by Walsham (1993) who identifies four forms of generalisation, all of which are met by this study:

- **Development of concepts:** The concept of time-based release management is elaborated and a number of associated concepts have been described and operationalised in relation to release management and coordination in large software projects;
- **Generation of theory:** The rich data gathered in this study about FOSS projects have been used to generate a theory of time-based release management;
- **Drawing of specific implications:** A number of implications follow from the theory that have practical value to the FOSS community, such as insights into factors influencing the choice of an appropriate release interval for a project;
- **Contribution of rich insight:** Since this research has gathered qualitative data from key personnel involved in release management, rich insights about the motives behind specific practices found in the FOSS community have been obtained. A good understanding of problems that can often be observed in FOSS projects and their causes has also been developed.

ANALYSIS OF FINDINGS

We discuss the research findings from each of the research phases below.

Phase 1 - Exploratory Interviews

As already mentioned, phase 1 of this study involved exploratory interviews with key developers and release managers of 20 FOSS projects (Table 3). The most significant findings from this phase relate to the identification of three different categories of release, to the preparation of stable releases, and to fundamental release strategies.

FOSS Release Categories

The three release categories identified differ quite significantly regarding the audience they

address and the effort required to deliver the release:

- 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 stabilised 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 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 written and the software needs to be packaged up.

Interestingly, development releases have become less important as developers are increasingly using version control systems to download the most recent version rather than relying on a development release that may be a few weeks out of date.

Preparation of Stable Releases

Preparing a stable release for end-users involves a complex set of tasks in which all developers on a project have to coordinate their work to deliver a high quality product. While the specific release approach may differ from project to project, we could identify a common pattern of staged progress towards a release where each stage is associated with increasing levels of control over the changes that are permitted. These control mechanisms are usually known as *freezes* since the development is slowly halted and eventually brought to a standstill:

- **Feature freeze:** No new functionality may be added. The focus is on the removal of defects;
- **String freeze:** No messages which are displayed by the program, such as error messages, may be changed. This allows translators to translate as many messages as possible to other languages before the release;
- **Code freeze:** Permission needs to be sought before making any change, even in order to fix bugs.

Among the twenty projects in phase 1, there was no common pattern as to how often new user releases are published. The release frequency ranged from one month to several years. A number of factors were identified which are related to the release frequency:

- Build time: some projects require massive processor power to compile a binary that can be executed and shipped to users. This long compilation step puts a natural limit on release frequency;
- Project complexity: projects consisting of many different components with large numbers of developers exhibit a tendency towards a slower release cycle due to the extra coordination burden;
- Age of the project: young projects tended to perform releases more frequently. This is largely because young projects need more direct feedback from users than already established projects. Also, young projects are likely to be smaller and thus it can be easier to prepare releases;
- Nature of the project: projects which are aimed at the desktop or other fast-paced environments have a much higher release frequency than software which is mainly

used on servers where there is a tendency to avoid upgrades unless they are strictly necessary. The audience also plays an important role. Projects which are mainly oriented towards developers or experienced users may make frequent releases because such users are often interested in the latest technology.

There is another factor which has a major impact on the whole release strategy of a FOSS project: the inclusion of the project's software in a collection of FOSS software, such as a Linux distribution. FOSS projects publish their work independently but for a complete system hundreds or even thousands of component applications are required. A number of non-profit projects and companies exist whose purpose it is to take individual FOSS applications and integrate them to a system which is easy to install and use. There are commercial companies which provide such integrated systems, such as Red Hat or Novell with their SUSE Linux, as well as non-profit organisations, such as Debian and Gentoo. The inclusion in such systems is seen as a very positive factor for a project because its software is thereby exposed to a much wider audience. At the same time, this greater volume of end-users often requires changes to the release strategy because the project is no longer solely used by developers who are inherently more familiar with the software.

Release Strategies

While there are many differences regarding the specific details of the implementation of a release management strategy, the following two fundamental strategies have been identified:

- Feature-based strategy: the basic premise is to perform a new release when a specific set of criteria has been fulfilled and certain goals attained, most typically a number of features which developers perceive as important. This strategy is in line with traditional software release management which is feature-driven;
- Time-based strategy: a specific date is set for the release well in advance and a schedule created so people can plan accordingly. Prior to the release, there is a cut-off date on which all features are evaluated regarding stability and maturity. A decision is then made as to whether they can be included in the release or whether they should be postponed to the following release.

A number of projects reported growing frustration with the feature-based release strategy which results in very ad-hoc processes. All functionality that is desired is never achieved and so the release manager has to call for a release at some point, often very unexpectedly. The previous release is typically quite dated and so there is a great rush to get a new release out of the door. This lack of planning can lead to incomplete features and insufficient testing. This strategy is also often associated with the motto "*release when it's ready*". Even though this is a laudable goal, in practice it is often problematic, particularly in large projects where there are always new features that could be added or bugs that could be fixed. This approach results in major delays because the project is constantly at the point where it could make a release but there is always something that remains to be done. In the meantime, the last stable release becomes increasingly out of date.

In order to address these problems about a quarter of the projects investigated in Phase 1 were considering a migration to a time-based strategy. In their view, time-based releases constituted a more planned and systematic approach to release management, making release coordination easier. This issue is the main subject of the phase 2 case study research.

Phase 2 - Case Studies

As already mentioned, phase 2 of the research involved case studies of seven FOSS projects

(Table 4). Here we present a cross-case analysis, initially focusing on the lack of planning and ad-hoc release management practices across projects, and the consequent negative short and long-term effects. We then focus on the implementation of a time-based release management strategy.

Ad-Hoc Management Processes

In distributed volunteer projects, the lack of release planning places extreme coordination overhead on the individuals responsible for release management. Extreme (and increasingly long) delays in releases had been common across several of the projects; indeed, Debian had achieved the unenviable reputation of never being on time. Due to lack of overall planning, instructions to prepare for a release usually came out of the blue. This resulted in a flurry of development activity as developers tried to make changes to be included in the next release. Rather than slowing down development, freeze announcements actually had the opposite effect, in what was aptly described by a Linux kernel developer as "*a thundering herd of patches*", and inevitably delaying the actual release:

"I think that freezes were sudden, and, like in Debian, we were promised a freeze and then it wouldn't happen for six months. This means six months of working incredibly hard for a deadline which is constantly moving away from you." (Murray Cumming, GNOME)

This also had an impact on vendors who needed to incorporate these releases as part of their distributions:

When you made your changes on the development branch you wouldn't know when you would be able to use those changes. But if you're making changes on the stable branch, you're changing very old code. (Havoc Pennington, Red Hat)

As a result, vendors tended to backport changes from the development version to the last stable release which led to much fragmentation. The situation was captured well in relation to OpenOffice.org:

It was very difficult to predict when it [2.0] would be ready, and as a consequence, we shipped a product based on release snapshots made three months before 2.0 while trying to bug fix those in parallel. (Michael Meeks, Novell)

In summary the above led to immediate problems in the short-term such as:

- Huge number of changes to test as developers added features indiscriminately;
- Little testing of development releases as these moved increasingly further from the latest stable release;
- Fragmentation of development as vendors chose to work with their own versions and avoid the official release;
- Out-of-date software due to long delays between stable releases.

These ad-hoc processes also led to long-term problems such as loss of credibility for the project, and fewer contributors for the project as developers become disillusioned with delays.

A Time-Based Release Management Strategy

All projects were moving towards a time-based release management strategy based on the early successful experiences of projects such as GNOME and GCC. However, four conditions appear to be essential to pursue a time-based release management strategy:

- **Sufficient development done in release interval:** While this may seem a very obvious pre-condition, many FOSS projects on forums such as SourceForge and Freshmeat show very little development activity, e.g. no change in version number or code size

over an extended time-period (Capiluppi et al., 2003; Howison & Crowston, 2004). Thus, time-based releases would not be relevant to many FOSS projects as insufficient development would have taken place;

- **Distribution costs cheap:** If releases are to be published and delivered at regular intervals, distribution must be inexpensive and easy. While distribution may be on CDs, increasingly releases are distributed via a web-site from which end-users and vendors can access the relevant releases;
- **Release rationale not driven by specific functionality:** Traditionally, software distributed as a shrink-wrapped product will tend towards providing new functionality to incentivise customers to upgrade. For FOSS projects it is important that they are not constrained to provide specific functionality in a release as such constraints may delay the project if development and testing are not completed before the release deadline. Rather, frequent releases tend to be welcomed in the FOSS context;
- **Modular code and project structure:** While this requirement relates to the code base and organisational structure of the project, in effect the two are highly correlated. Indeed, large FOSS projects have been shown to be an aggregation of smaller projects (Crowston & Howison, 2005). If components in a release are modular, then any defective modules can be swapped out of the release, as components can be developed, fixed and released more independently. This insight is very important in terms of time-based release management because it allows the implementation of two complementary release mechanisms: individual components may be 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.

Time-based release management also helps address coordination mechanisms as discussed earlier. For example, instead of active task assignment, FOSS relies on self-assignment of tasks. Development in FOSS projects is done in a massively parallel way as individual developers work independently on features they are interested in – the principle of *optimistic concurrency*. This self-selection mechanism works especially well in large projects as it allows developers to work in areas in which their expertise is best suited. Coordination then takes place after the fact when the best solutions are chosen (Yamauchi et al., 2000).

An important coordination mechanism that also arises from time-based releases is that of a *regular schedule*. The objective of time-based releases is to announce a target date well in advance and then publish a schedule with important milestones leading to the target date. A regular schedule creates a number of significant benefits, discussed in turn below:

Provides a Regular Reference Point

The parallel and independent nature of FOSS development reduces the amount of active coordination needed. However, regular synchronisation is important so that developers become aware of other changes that may conflict with their own work or have other important implications. GNOME developer Jeff Waugh suggests a useful analogy with MPEG video compression. Such compression algorithms do not store each picture frame individually. Instead, they store one frame and subsequent changes made to that frame. At some point, they include a full frame again and then record only changes made to this frame. This mechanism reduces the amount of storage space because not every frame is stored as an entire frame. At the same time, it provides a safety mechanism to allow reconstruction because it periodically stores a full frame from which to start again. This frame, which contains the entire screen, is known as the key-frame. Waugh argues that regular releases act as a key-frame:

For us, the stable release is the key-frame — a full complete picture of where we are. Development is the modification to the key-frame. Then you have another key-frame — the full picture. There are only certain things changing, you're never unclear about what has changed, you know what needs to be tested.

Promotes Developer Discipline and Self-Restraint

One of the negative aspects of irregular feature-based releases is that developers rush to get their work included as they do not know when the next release will take place. When a regular schedule is in place, it is easier to persuade developers to revert features from the release if things are not working smoothly.

Improved Familiarity with the Process

If releases happen infrequently, developers and release managers are less sure about the process. There is much uncertainty and fire-fighting, and problems inevitably occur. By implementing a regular release cycle, developers become more accustomed to it. Also, this familiarity helps reduce the burden on the release manager as developers learn to coordinate better through growing familiarity with the process.

Self-Policing of Simultaneity Constraints

Simultaneity constraints were identified as an important coordination dependency earlier. However, a published schedule allows this to be self-policing, thus reducing the active coordination required by the release manager. The schedule becomes the overall planning tool to define interdependencies between activities. This establishes deadlines for different activities and arranges activities in a natural order. This is crucial for many tasks and individuals – translators, for example, who can only perform their work when the documentation they need to translate has been finished and is no longer in a state of flux. The schedule not only tells translators when they can start their work, but by specifying a 'string freeze' it will also tell developers when they must stop making changes to texts ('strings').

A clear schedule also allows a vendor to participate more closely in the development of the project. With the help of a schedule, vendors can decide whether new functionality they would like to ship should be developed as part of the official project or be part of their own development line. The predictability offered by time-based releases encourages vendors to work on the official project and decreases fragmentation, which has often occurred in FOSS projects in the past.

Creation of a Release Schedule

Given the extent to which the release schedule acts as a coordination mechanism, it needs to be carefully planned. A necessary first step is to choose the release interval. Broadly speaking, it is important to strike a good balance between leaving enough time to develop new features on the one hand, and to perform testing and release preparations on the other hand. One of the main criteria a schedule has to fulfil is to be realistic. While the majority of developers are primarily interested in adding as many new features as they can, the project as a whole, led by the release manager or core developer, has to be realistic as to how much new code can be added so that it can still be sufficiently tested within one release interval.

At a higher level also, there are significant 'network effect' advantages to be gained if a project can synchronise its release schedule with those of other projects from which it may leverage benefits. For example, one of the key reasons why the Plone project has decided to move to a six-month time-based release strategy was to align its development closer with that of Zope. Plone is built on top of Zope and the implementation of a similar release strategy will allow the project to use the newest technologies developed by Zope.

In keeping with this, a large number of time-based FOSS projects have chosen a release interval of six months. Since major

Linux distributions, such as Fedora, follow the same release interval this ensures that these distributions will be able to ship the latest releases produced by many FOSS projects. This increases the exposure of the software and may lead to better feedback. It may also provide a further incentive for vendors to get involved in important projects and help them meet their release targets, as their own releases might otherwise become jeopardised.

Finally, at a high level, given the voluntary nature of FOSS contributions, releases should be avoided during holiday periods, and given the global nature of FOSS development this extends to holidays relevant to all cultures and traditions

Following the choice of release interval, a necessary next step is the identification of dependencies, the essence of which is captured in the following:

We have dependencies. Applications depend on APIs, translations depend on strings, documentation depends on the UI [user interface], the UI depends on application writers and the API. (Murray Cumming, GNOME)

Finally, the granularity of the schedule needs to be planned. The release interval must be divided into different phases such as development and testing. Experience from previous releases should be factored in as well as dependency information.

SUMMARY AND CONCLUSION

Here we briefly summarise the findings before discussing the implications of theory and practice.

Summary of Findings

Motivated by the continuing maturation of FOSS towards more hybrid commercial forms, this research focused on how quality issues such as quality and sustainability could be improved in FOSS. Coordination in software development generally is a critical issue (Herbsleb & Mockus, 2003). This is further exacerbated in distributed development contexts, and even more so when the majority of developers are volunteers, as is the case with FOSS projects. Given that stresses come to a head during product releases, release management is clearly a topic which should be researched in some detail.

Overall, our research found that the feature-based release strategy common to traditional software development often causes problems in relation to FOSS coordination and planning, and results in delays, lack of testing, reduced motivation of developers, fragmentation of development as vendors created their own versions rather than relying on the official release, and overall loss of credibility for the project.

Time-based release management, on the other hand, reduces the amount of active coordination required because it allows developers to work with greater independence. Also, it allows projects to concentrate their resources on creating infrastructure and mechanisms to support collaboration an coordination around the critical time of a release. This serves to keep participants informed about the status of the project, and helps increase trust in the process and motivates contributors to participate in the release process.

Our main findings are summarised in Table 5.

Implications

This study has a number of theoretical and practical implications. We have provided several examples where time-based release management acts as a significant coordination mechanism. Table 6 illustrates how we build upon the coordination theory of Malone and Crowston's (1993) coordination concepts as we have particularised their framework in relation to FOSS release management.

With regards to practical implications, much of the advice to FOSS practitioners is summarised in Table 6. However, a number of other issues could be explored to good effect.

Table 5. Summary of findings on time-based release management in FOSS

FOSS Time-Based Release Management		
Preconditions	**Benefits**	**Creating a Schedule**
• Sufficient development done in release interval • Distribution costs cheap • Release rationale not driven by specific functionality • Modular code and project structure	• Provides a regular reference point • Promotes developer discipline and self-restraint • Increased familiarity with process reducing release manager burden • Self-policing of simultaneity constraints	• Choose release interval o Balance between what is realistic and desirable to achieve o Possible network effects from synchronisation with other projects o Identify periods to be avoided • Identification of dependencies • Plan granularity of schedule (development, testing etc)

Table 6. Activity dependencies and coordination processes

Dependencies	Coordination Process	Relevance to FOSS Time-Based Release Context
Shared resources	First-come, first-served; priority order, budgets, managerial decision, market-like bidding	Optimistic concurrency – parallel development Coordination after the fact
Producer-consumer relationships • Prerequisite constraints • Transfer • Usability	Notification, sequencing, tracking Inventory management Standardisation, ask users, participatory design	Published schedule
Simultaneity constraints	Scheduling, synchronisation	String freezes Release reference points
Tasks and subtasks	Goal selection, task decomposition	Self-selection of tasks Identification of dependencies

In the GNOME project, German (2004) found that paid employees were responsible for certain less-attractive tasks such as testing and documentation, which did not attract the attention of volunteers. Given the increased commercial involvement in FOSS, it would be interesting to assess the effect of professionalising and compensating the release management role and related tasks, on the basis that these are key tasks rather than being inherently unattractive.

The length of the release cycle is obviously an issue which requires balancing and will vary across FOSS projects and probably also according to project maturity, as younger projects will probably release more often to get feedback etc. Too long a release interval may reduce motivation levels and give the impression of a moribund project. On the other hand, too frequent a release schedule may limit radical innovation and ambition as only functionality that can be accomplished in a release interval may be considered for implementation.

This work could be specifically extended empirically and quantitatively to establish whether time-based releases lead to higher levels of motivation among developers, or what impact it has on the level of code contributions. Also, it would be interesting to test whether the amount and quality of feedback is higher in time-based release situations.

Moving to the software industry more generally, the trend towards software as a service also suggests that a big-bang feature-based release management strategy is not well-suited as customers are more likely to appreciate

continuous improvements. In this scenario, customers will use the latest software from a vendor web-site rather than buy a new shrink-wrapped product. Thus, regular additions of new functionality and a healthy metabolism of a product under active development will be more appropriate, and in turn this calls for a time-based release management strategy.

Moving beyond the software domain, in other contexts involving voluntary contributions, it would be useful to assess the extent to which the lessons from this study are applicable to other contexts.

Finally, to return to the metabolism metaphor, one of the interviewees referred to the 'pulse of a project' which is determined by its release activity. It certainly seems that time-based release management contributes greatly to a healthy pulse.

ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie). Martin Michlmayr is currently at Hewlett-Packard.

REFERENCES

Baetjer, H. (1997). *Software as capital*. Los Alamitos, CA: IEEE Computer Society Press.

Baskerville, R., Levine, L., Pries-Heje, J., Ramesh, B., & Slaughter, S. (2002). Balancing quality and agility in internet speed software development. In *Proceedings of the International Conference on Information Systems* (ICIS), Barcelona.

Bergquist, M., & Ljungberg, J. (2004). The power of gifts: Organising social relationships in open source communities. *Information Systems Journal*, *11*(4), 305–320. doi:10.1046/j.1365-2575.2001.00111.x.

Boehm, B. B. (1981). *Software engineering economics*. Upper Saddle River, NJ: Prentice Hall.

Bogdan, R., & Taylor, S. (1975). *Introduction to qualitative research methods*. New York, NY: Wiley & Sons.

Brooks, F. (1987, April, 10-19). No silver bullet: Essence and accidents of software engineering. In H.-J. Kugler (Ed.), *Proceedings of the IFIP Tenth World Computing Conference.*

Capiluppi, A., Lago, P., & Morisio, M. (2003). Evidences in the evolution of OS projects through changelog analyses. In *Proceedings of the 3rd Workshop on Open Source Software Engineering* (pp. 19-24). ICSE.

Corbin, J., & Strauss, A. (1990). *Basics of qualitative research: Grounded theory procedures and techniques*. Thousand Oaks, CA: Sage.

Crowston, K. (1997). A coordination theory approach to organizational process design. *Organization Science*, *8*(2), 157–175. doi:10.1287/orsc.8.2.157.

Crowston, K., Howison, J. (2005). The social structure of Free and Open Source software development. *First Monday*, *10*(2).

Dayani-Fard, H., Glasgow, J., & Mylopoulos, J. (2005). A datawarehouse for managing commercial software release. In *Proceedings of the 21st International Conference on Software Maintenance*, Budapest, Hungary.

Du, G., & Ruhe, G. (2005). Identification of question types and answer types for an explanation component in software release planning. In *Proceedings of the Of K-CAP 2005,* Banff, Canada (pp. 193-195).

Erdogmus, H. (1999) Comparative evaluation of software development strategies based on net present value. In *Proceedings of 1st International Conference on Economics-Driven Software Engineering Research,* Toronto. Canada.

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

Fitzgerald, B. (2006). The transformation of open source software. *Management Information Systems Quarterly*, *30*(3), 587–598.

Garzarelli, G., & Galoppini, R. (2003, November). Capability coordination in modular organization: Voluntary FS/OSS production and the case of Debian GNU/Linux.

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. doi:10.1002/spip.189.

Ghosh, R. A. (2006). *Economic impact of open source software on innovation and the competitiveness of the information and communication technologies (ICT) sector in the EU (Tech. rep.). Maastricht Economic and Social Research and Training Centre on Innovation and Technology*. The Netherlands: United Nations University.

Glaser, B., & Strauss, A. (1967). *The discovery of grounded theory: Strategies for qualitative research*. Chicago, IL: Aldine.

Glass, R. (1991). *Software conflict: Essays on the art and science of software engineering*. Englewood Cliffs, NJ: Prentice Hall.

González-Barahona, J. M., Ortuño Pérez, M. A., de las Heras Quirós, P., Centeno González, J., & Matellán Olivera, V. (2001, December). Counting potatoes: The size of Debian 2.2. *Upgrade*, *II*(6), 60–66.

Greer, D., & Ruhe, G. (2004). Software release planning: An evolutionary and iterative approach. *Information and Software Technology*, *46*, 243–253. doi:10.1016/j.infsof.2003.07.002.

Herbsleb, J. D., & Mockus, A. (2003). Formulation and preliminary test of an empirical theory of coordination in software engineering. In *Proceedings of the 9th European Software Engineering Conference*, Helsinki, Finland (pp. 138-147).

Howison, J., & Crowston, K. (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).

Krishnamurthy, S. (2002). Cave or community? An empirical examination of 100 mature open source projects. *First Monday*, *7*(6).

Lee, A. S., & Baskerville, R. L. (2003). Generalizing generalizability in information systems research. *Information Systems Research*, *14*(3), 221–243. doi:10.1287/isre.14.3.221.16560.

Levin, K. D., & Yadid, O. (1990). Optimal release time of improved versions of software packages. *Information and Software Technology*, *32*(1), 65–70. doi:10.1016/0950-5849(90)90048-V.

Li, P. L., Shaw, M., & Herbsleb, J. (2003). Selecting a defect prediction model for maintenance resource planning and software insurance. In *Proceedings of 5th Int'l Workshop on Economics-Driven Software Engineering Research*.

Malone, T. W., & Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Computing Surveys*, *26*(1), 87–119. doi:10.1145/174666.174668.

Malone, T. W., Crowston, K., Lee, J., & Pentland, B. (1993). Tools for inventing organizations: Toward a handbook of organizational processes. In *Proceedings of the Second IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises* (pp. 72-82).

McLean, E. (1973). Empirical studies of management information systems. In R. Van Horn (Ed.), *DataBase, winter* (pp. 172–180).

Michlmayr, M., & Hill, B. M. (2003). Quality and the reliance on individuals in free software projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, OR, (pp. 105-109). ICSE.

Michlmayr, M., Hunt, F., & Probert, D. (2007). Release management in free software projects: Practices and problems. In J. Feller, B. Fitzgerald, W. Scacchi, & A. Silitti (Eds.), *Open source development, adoption and innovation*, (pp. 295-300). International Federation for Information Processing: Springer.

Miles, M. B., & Huberman, A. M. (1994). *Qualitative data analysis*. Thousand Oaks, CA: Sage Publications.

Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, *11*(3), 309–346. doi:10.1145/567793.567795.

Raymond, E. S. (1999). *The cathedral and the bazaar*. Sebastopol, CA: O'Reilly & Associates.

Robbins, J. (2002). Adopting OSS methods by adopting OSS tools. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*, Orlando, FL.

Ruhe, G., & Greer, D. (2003). Quantitative studies in software release planning under risk and resource constraints. In *Proceedings of the IEEE-ACM International Symposium on Empirical Software Engineering* (pp. 262-271).

Sassenburg, H., & Bergout, E. (2006). Optimal release time- numbers or intuition? In *Proceedings of WoSQ*, Shanghai, China.

Smith, N. (1990). The case study: a useful research method for information management. *Journal of Information Technology*, *5*, 123–133. doi:10.1057/jit.1990.30.

Tawileh, A., Rana, O., Ivins, W., & McIntosh, S. (2006). Managing quality in the free and open source software community. In *Proceedings of the 12th Americas Conference on Information Systems*, Acapulco, Mexico.

Walker, R. (1988). *Applied qualitative research*. Farnham, UK: Gower.

Walsham, G. (1993). *Interpreting information systems in organizations*. Chichester, UK: John Wiley & Sons.

Yamauchi, Y., Yokozawa, M., Shinohara, T., & Ishida, T. (2000). Collaboration with lean media: How open-source software succeeds. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, Philadelphia, PA (pp. 329-338).

Yin, R. K. (1994). *Case study research: Design and methods*. Thousand Oaks, CA: Sage Publications.

Zhao, L., & Elbaum, S. (2003). Quality assurance under the open source development model. *Journal of Systems and Software*, *66*, 65–75. doi:10.1016/S0164-1212(02)00064-X.

Martin Michlmayr works as an Open Source Community Expert for HP's Open Source Program Office. In his role, he facilitates open source activities both internally within HP as well as externally within the broader open source community. Michlmayr has been involved in various free and open source software projects for over 15 years. Prior to his work at HP, he was a volunteer coordinator for the GNUstep Project, acted as publicity director for Linux International and served as the leader of the Debian project. In the two years as the leader of Debian, he performed important organizational and coordination tasks. Michlmayr currently serves on the board of the Open Source Initiative (OSI), the non-profit organization that maintains the Open Source Definition. Michlmayr holds Master degrees in Philosophy, Psychology and Software Engineering. He earned a PhD from the University of Cambridge.

Brian Fitzgerald holds an endowed professorship, the Frederick A Krehbiel II Chair in Innovation in Global Business & Technology, at the University of Limerick, Ireland, where he has also been Vice President Research from 2008-2011. He is Principal Investigator in Lero - the Irish Software Engineering Research Centre, and was Founding Director of the Lero Graduate School in Software Engineering. He was formerly at University College Cork, and has held visiting positions in Italy, Austria, Sweden, US and the UK. He holds a PhD from the University of London and his research interests lie primarily in software development, encompassing development methods, global software development, agile methods and open source software. His publications include 12 books, and over 130 peer-reviewed articles in the leading international journals in both the Information Systems and Software Engineering fields.

APPENDIX

Table 7. Quotes from research interviews

Quote from Research Interviews	Analysis
My impression is that for almost two years people kept asking, "is the API stable yet?" or, "have you finished with this interface, can we start translating it now?"	Lack of communication leads to a dependence on the release manager and to a more centralised development process.
You have the problem that you cannot suddenly say that now we have a freeze. For people really to be prepared for it, they need to know, I think, several months in advance what is going to happen.	Individual developers require information about the release in order to perform their work.
I think part of it was that [a 6 month cycle] gives you enough time to develop some new features without too much time to get too far away from the previous version.	The release cycle has to find a compromise between conflicting interests (e.g. doing more development vs performing a release).
When you're doing a time-based release, all you ever have to say is that if you revert it you can put it into the next development phase.	Having a clear schedule allows better control over developer input.