Balancing Technological and Community Interest: The Case of Changing a Large Open Source Software System

Abstract. This paper studies the process of rewriting and replacing critical parts of a large open source software (OSS) system. Building upon the notions of installed based and transition strategies, we analyze how the interaction between the OSS and the context within which it is developed and used enables and constrains the process of rewriting and replacement. We show how the transition strategy emerges from and continuously changes in response to the way the installed base is cultivated. By demonstrating a mutual relationship between the transition strategy and the installed base, we show how the transition strategy in this particular case changes along three axes: the scope of the rewrite, the sequence to replace existing software, and the actors to be involved in the process. The paper is concluded with some implications for how to study the process of rewriting and replacing OSS.

Keywords. Open source software development. Rewrite and replacement. Transition strategy. Installed base.

Introduction

Parallel development, a rapid release schedule, actively involved users, and prompt feedback are described as key characteristics of open source software (OSS) development (Feller & Fitzgerald 2002). Empirical studies of OSS development have therefore primarily focused on the cyclic process of corrective and adaptive maintenance (German 2005), its organization (Crowston & Howison 2005), and analysis of the products of this process (Paulson et al. 2004, Samoladas et al. 2004, Mockus et al. 2002). Describing the process of rewriting the FreeBSD kernel, Jørgensen (2001) shows that unlike the discretely delineated

tasks of adaptive and corrective maintenance, rewriting OSS is a longitudinal process that does not lend itself well to parallel development, rapid release schedule, and active user involvement. While we know that large and successful OSS products are rewritten–for instance the original Apache code was rewritten and replaced with a modular design in 1995, and several large subsystems of the Linux kernel, like virtual memory handling, have been rewritten and replaced throughout the kernel's life cycle–we find that rewriting and replacing is an underdeveloped topic within OSS research.

Building upon Jørgensen's (2001) work, we study the repeated attempts at rewriting and replacing a core OSS system. The empirical basis for this paper is a study of the PenguinOS Linux distribution (PenguinOS is a pseudonym to anonymize the case). The background for the study is that the PenguinOS package manager, the core of the PenguinOS Linux distribution "is very fragile [because it has] evolved rather than being designed", as one of the PenguinOS developers puts it. Studying the attempts at rewriting and replacing the package manager provides an excellent opportunity to study the problems associated with rewriting and replacing critical parts of a large OSS system. To this end, we ask: How does the interaction between the OSS and the context within which it is developed and used enable and constrain the process of rewriting and replacement? In this paper we analyse this by studying the relationship between the installed base and transition strategies (Hanseth and Monteiro 1998) in the process of rewriting and replacing the PenguinOS package manager.

The remainder of the paper is structured as follows. The second section motivates the study of rewriting and replacing OSS through the notions of transition strategies and installed base. These two terms are elaborated. The third section outlines the case; presenting the research setting, as well as describing three attempts at rewriting and replacing the package manager. In the fourth section we discuss the case along two dimensions that surface in the case: the issue of resources and transition strategies as a process. The final section contains concluding remarks, where we describe how we have addressed the research question and implications of our findings to the study of rewriting and replacing OSS.

Methodologically, the paper is based on an interpretive case study (Klein & Myers 1999) of the PenguinOS OSS community. The data was primarily collected during a ten months programme of participant-observation conducted from March to December 2004. Since the OSS community is geographically distributed, participant-observation took the form of observing and participating on the Internet Relay Channels (IRC) that the community use for communication, by submitting and resolving failure reports, as well as contributing with code. Throughout the period of fieldwork the IRC channels we participated on were logged to disk; one file each day for each IRC channel totalling 1027 files. A key informant also provided us with his IRC logs, stretching back to April 2003. No

formal interviews of participants in the OSS community were undertaken, although informal talks with participants–both on e-mail and on IRC–were conducted on a regular basis to test our informal theories about the fieldwork. 71 documents were collected throughout the period and organized in a documentary database. Online data sources that provide static data were surveyed. These include the PenguinOS bug tracking database, the PenguinOS mailing list archives, and the PenguinOS revision control system. As the PenguinOS Web site is under revision control, relevant documents from this Web site were not organized in the documentary database. Instead, we decided to rely on PenguinOS's revision control system. This archival material provided us with data from 2002 to the end of 2005. A more thorough presentation of the research is provided in (reference undisclosed for reviewing purposes).

Theory

Jørgesen (2001) describes the process of implementing symmetric multi-processing, a significant new feature, in the FreeBSD operating system kernel. Although the paper describes in detail the practical arrangements for making the significant change and folding it into the main code base, the paper tells little about the context and rationale for organising the process this way. However, the paper provides little information about how the OSS developers decide upon the specifics of this process of going from one version of the software to other. We expand upon Jørgensen's (ibid.) work, by examining how OSS developers make such decisions. We do so by analysing the OSS an information infrastructure (II) (Hanseth and Monteiro 1998), studying the process of rewriting and replacing the PenguinOS package manager in terms of transition strategies and installed base.

Transition strategies

The transition strategy is a plan outlining how to go from one stage of the II to the other (Monteiro 1998). However, the transition strategy is caught in a dilemma, "where the pressure for making changes … has to be pragmatically negotiated against the conservative forces of the economical, technical, and organizational investments in the … installed base" (ibid., p. 230). Controversies over a transition strategy are therefore negotiations about *how* big changes can–or have to– be made, *where* to make them, and *when* and in *which* sequence to deploy them.

Whereas Jørgensen (2001) describes the sequencing when rewriting a clearly delineated part of the software, thinking in terms of transition strategies enables us to study the larger process of rewriting software encompassing what is to be rewritten and the scope of the changes, important factors in the process of rewriting the PenguinOS package manager.

Installed base

The installed base can be defined as the interconnected technologies and practices that are institutionalised in an organization (Hanseth and Monteiro 1998). Adopting this view, we see that changes cannot be made to software artefacts in isolation, but must always take into account the other elements of the installed base that the artefact is connected to.

This points towards two important elements when thinking in terms of installed base. One, II's must evolve by extending and improving the existing installed base, or *cultivating the installed base* as it is called (ibid.). Two, as II's grow, it becomes increasingly hard to extend and improve it because of the many elements that have to be changed in the process. This is called the *inertia of the installed base* (ibid.).

Actor-network theory

Like II, actor-network theory (ANT) is the underlying ontology for this study as well. We therefore mobilise a limited ANT vocabulary inscribed in and circulated by Callon (1986) and Latour (1987) for the case description and analysis of this paper. Well aware of recent movement toward fluids and fiery objects both within ANT and IS research, we choose to mobilise this vocabulary as it translates well our interest in bringing forth the chronic tension of multiple and at times contradictory interest in cultivating the PenguinOS installed base.

A major focus of ANT is to provide a way of tracing and explaining the process of how networks of actors, *actor networks*, become more or less stable through the alignment of interest. Particular to ANT is that the notion of actors encompasses both human and non-human actors such as software technologies, documents, and so on.

The process wherein networks of aligned interest are created and maintained, is called *translation*. Through the process of translation the translating actor defines other actors, endowing them with interests and problems to be overcome. By framing a problem in such a way that it determines a set of actors, the translating actor defines and aligns the other actors' interests with his own (Callon 1986). The problem is framed in to establish the translating actor as an *obligatory passing point* by enrolling and mobilising the other actors to pass through this point to achieve their interests.

Translation is therefore the process of enrolling a sufficient body of actors by aligning these actors' interests so that they are willing to participate in particular ways of acting. It implies definition, and this definition is *inscribed* in material intermediaries (Latour 1986). These intermediaries are actors in their own right. They are delegates who stand in for and speak for particular interests; they are the medium in which interests are inscribed. The operation or translation is therefore

triangular: it involves a translating actor, actors that are translated, and a medium in which the translation is inscribed.

The Case of Rewriting and Replacing PackMan

GNU/Linux distributions, complete operating systems that integrate the Linux operating system kernel with a collection of software libraries and applications, are an intrinsic part of the success of Linux. Since the beginning of the Linux kernel development in the early 1990s, communities of OSS developers have created GNU/Linux distributions. As GNU/Linux distribution consists of thousands of different software libraries and applications, distribution developers primarily repackage third-party OSS, doing whatever adaptations required for the third-party software to function on their specific GNU/Linux distribution. At the time of writing, there are over 300 Linux distributions, large and small–some developed commercially, others developed by volunteers–registered with the DistroWatch (2006) Web site. In this paper we report from a study of the OSS community developing the PenguinOS Linux distribution, rated by DistroWatch among the ten most widely used distributions.

Starting out as a one-man volunteer project in 2000, by 2003 the number of volunteer PenguinOS developers had grown to over 200. The number of third-party software libraries and applications, collectively labelled packages, supported by the PenguinOS Linux distribution had also grown. From being a GNU/Linux distribution, PenguinOS had over time been turned into a generalized software system for distributing OSS software packages for different Unix operating systems like BSD and MacOS. By 2003 PenguinOS suffered increasingly from growth pains.

Organizationally, they PenguinOS developers addressed the growth pains by introducing a formal management structure in June 2003: "The purpose of the new management structure is to solve chronic management, coordination and communication issues in the PenguinOS project" (reference undisclosed for reviewing purposes). Technically, by mid-2003 growth pains were putting a strain on the PenguinOS package manager, PackMan, the software that integrates packages on local PenguinOS systems. It is from the repeated attempts at rewriting and replacing the package manager that we report in this paper. Although all of the PenguinOS developers can agree that the package manager needs to be rewritten and replaced, this turns out to be problematic. After numerous attempts, the PenguinOS developers give up. Why is it that they fail to rewrite and replace the package manager? We provide an overview of these attempts in the rest of this section, before we address the above question during the discussion in section 4.

First attempt

It is mid-November 2003. Four developers make a forceful declaration of intent during the biweekly PenguinOS managers' meeting: "We are aggressively working on plans for next generation PackMan, which is not going to simply be a rewrite or a new version but beyond people's wildest expectations". The source code of the current version of PackMan "is very fragile [because it has] evolved rather than being designed". It has become difficult to comprehend and maintain, preventing the PenguinOS developer community at large from participating in developing and maintaining the package manager. Currently, only a "small group [of PenguinOS developers] really know how to make significant contributions to the code".

To enrol the PenguinOS developer community with the rewrite effort, the four developers provide an architecture diagram (see Figure 1). The diagram graphically lays out the main parts of the package manager, the interface between these parts of the system, and which features will be supported as components.

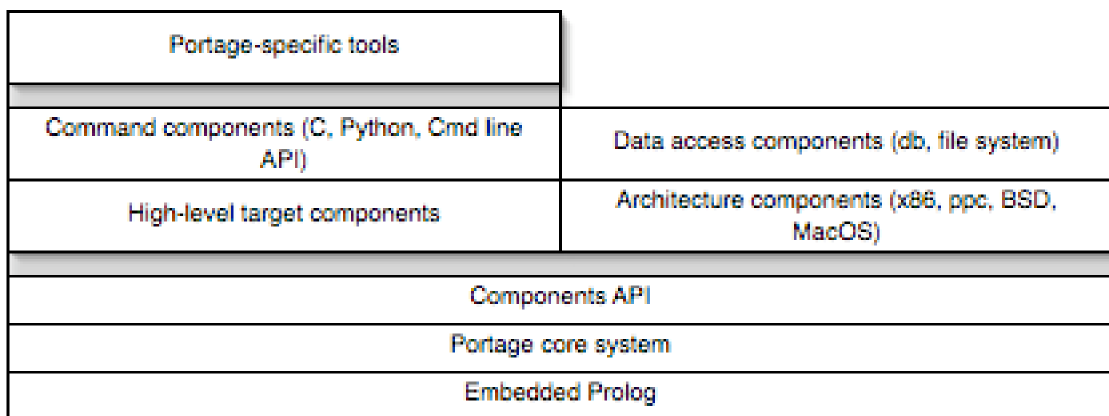

Figure 1 PackMan-ng architecture diagram

By rewriting PackMan with a core system and "a solid API for components [where] major parts that are now core PackMan are going to be implemented as components", the four developers explain, "components can be developed by different teams [of PenguinOS developers", turning PackMan into "a true community project". To achieve this end, they continue, PackMan "is not just to be 'robust enough' but incredibly reliable".

The architecture diagram serves to meet the interests of two other actors. Performance of the package manager has been a point of discontent among the people administrating PenguinOS systems. Furthermore, a number of PackMan-specific applications that are part of the PenguinOS software distribution operate directly on PackMan's database and configuration files. A recurring problem with changing the format of these configuration files and databases, is that some of the

PackMan-specific tools cease to function. To meet these interests, the four developers are developing a prototype of the core system.

The prototype is realized in GNU Prolog, as this programming language can meet the above interests. Prolog can provide "robust, provably correct code". GNU Prolog has an API for components to be written "in C for performance when needed". However, the final choice of realization language is to grow out of the requirements. "Right now," the four developers explain, "we are at the blueprint stage … the plan is to get a solid blueprint, then make it a community project at the earliest possible point". While the four develop the blueprint and the prototype, they enrol the PenguinOS developer community at large to formulate requirements for the rewrite.

No one in the community questions the rationale for rewriting PackMan from scratch with a modular architecture. However, the choice of Prolog for a prototype produces resistance. How can Prolog resolve the problem of performance, when "Prolog could be very slow"? one developer asks. Also, how can PackMan be turned into a true community project when only very few PenguinOS developers are familiar with the predicate-logic programming paradigm of Prolog? The choice of realization language will produce a high entry-barrier, some developers argue.

The promised Prolog prototype fails to manifest, and in mid-December 2003 a competing prototype realized in Ada appears. Throughout November and December the four developers planning to rewrite PackMan keep on trying to enrol the PenguinOS developer community with their plan by pointing out time and again that the choice of realization language is to emerge from the requirements. However, instead of formulating requirements, the PenguinOS developer community delve into endless discussions about the best programming language for rewriting PackMan.

By February 2004 all activities on this attempt to rewrite PackMan have ceased.

Second attempt

On February 18 2004 a new CVS module called PackMan-mod is imported into the PenguinOS CVS repository with the following note attached: "All current work between me and George moved from remote cvs to PenguinOS cvs!". Where PackMan-ng is a complete rewrite of PackMan from scratch, PackMan-mod is an effort to take the existing PackMan code and modularize it. Niles, a PenguinOS developer, is heading the effort with help from George, a newcomer to PenguinOS and not yet an official PenguinOS developer.

While Niles is modularizing the existing PackMan source code, George will help writing unit tests. According to the README file imported with the CVS module, the plan is that the "[d]evelopment of a package structure should

facilitate the later development of an consistent PackMan API, development of this API is part of this project and development should … begin once PackMan modularization is done and a unit testing framework is done."

Development on PackMan_mod is undertaken in parallel with the continued development and maintenance of PackMan. When the code is modularized, the plan is to rework changes made to PackMan during the period of modularization into the modularized version. However, it turns out that the changes made are too significant to achieve this, and this second attempt at rewriting and replacing PackMan is laid to rest.

Interlude

"I have a feature request for you", Bob states on the PackMan developers' IRC channel. It is mid-April 2004. Bob is a newcomer to the PenguinOS community, having only recently been adopted by the PenguinOS community to introduce web application support for PenguinOS. "The configuration tool for web applications need to edit the PackMan database," he continues, "so that a single web application may be installed multiple times on different locations in the file system. " The PackMan developers cannot see the purpose of such functionality. A discussion ensues. In the end Bob argues that if the PackMan developers cannot provide this functionality for him, he cannot provide support for web applications in PenguinOS. Reluctantly the PackMan developers agree with Bob about a technical solution to address his requirements.

Third attempt

In wake of the second attempt at rewriting PackMan, the remaining developer from that effort sets out to write an API on top of the existing implementation of PackMan. There is unanimous support for this effort among the other PenguinOS developers. The effort, while a continuation of parts of the second attempt at rewriting PackMan, also enrols the interests of two other developers who have been working to establish an API to insulate PackMan-specific applications from PackMan's configuration files and databases. This will solve the recurring problem of these applications breaking when the format of the configuration files and databases are changed. Furthermore, the API will insulate the core functionality of PackMan, so that after the API is in place modularization of PackMan may find place without disrupting users.

Work on this third attempt at rewriting PackMan ceases after a month and a half. The developer working on the API explains the situation:

> The whole API was designed around a single using application [that] would instigate the reading of the configuration, etc. … that doesn't fit in at all with distributed computing and/or remote management [which is something] people will ask for and/or want to implement

> themselves down the track. [It is therefore] better to preempt it now than find we've shot ourselves in the foot later.

The new approach for PackMan is to completely rewrite it with a core running as a Unix daemon with user applications calling the daemon remotely.

Upon the first author ending the fieldwork in December 2004, there are two independent efforts at rewriting PackMan. One effort by a young engineering student who has rewritten the core PackMan functionality in C, who fails to attract the PackMan developers' attention. Another effort by one of the PackMan developers to use experience from PackMan to write an independent package manager. This, he specifies, is "not a PackMan killer, but rather an independent implementation". However, in the future, his package manager may come to replace PackMan. As of writing this paper in November 2006, a new version of PackMan 2.0.51 is released, being simply the same code as in 2003 only with bug fixes and feature enhancements.

Although all of the PenguinOS developers can agree that the package manager needs to be rewritten and replaced, after numerous attempts they give up. Why is it that they fail to rewrite and replace the package manager?

Discussion

A number of problems are raised in connection with rewriting PackMan. Complex interdependencies between both modules and functions within the software makes it is difficult to understand parts of the software without a complete understanding of the whole. Interdependencies also make it difficult to make changes without breaking existing functionality. Because of this, only four PenguinOS developers know the source code well enough to make changes. Combined with the recurring problems of third-party applications, many of which operate directly on PackMan's different data bases with their proprietary data structures, ceasing to function after changes have been made to PackMan, the number of developers who can make meaningful changes to PackMan limits its continued development and maintenance of PackMan.

This is the situation that the PenguinOS developers time and again present and draw upon for motivating and explaining the interests and interest groups for rewriting the PackMan code and to justify their suggested solutions. The texture of the situation remains largely unchanged throughout the period. The problems they frame and the interests the PenguinOS developers construct all emerge from this context. In this section we will look closer at how this context enables and constrain the process of rewriting and replacing PackMan.

Mobilizing resources, balancing interests

Why do the repeated attempts at rewriting PackMan fail? Towards the end of April 2004, the PenguinOS developers describe the first attempt at rewriting PackMan as "hot air", "vaporware", and "mostly a buzzword". A predominant explanation for the repeated failures is exemplified by the following quote:

> A rewrite is a MAJOR waste of extremely limited resources. Unless PenguinOS gets MANY more PackMan devs OR can manage without a PackMan update for 6-12 months, a rewrite won't happen in any reasonable time … In the mean time, what happens with the existing implementation? Do you [have people] work on it? Or do you let it sit idle/stagnant. The amount of time it'd take would really drag out on the developers that want new features and simplifications … Resources are why the rewrites failed.

The issue of limited resources is the recurring explanation. The demise of both next generation PackMan and PackMan modularized are explained in terms of the strain on developer resources. However, given the number of PenguinOS developers, the programming resources within the community are significant. It is these resources the next generation PackMan developers want to tap in by turning PackMan into "a community project". It is therefore not because resources themselves are scarce that the rewrite efforts fail. The problem facing those who want to rewrite PackMan can be framed by Glass (1999, p.104)'s befuddlement: "I don't know who these crazy people are who want to write, read and even revise all that code without being paid anything for it at all." Similarly, based on the observation that the interests, needs, and know-how of OSS community members varies greatly, Bonaccorsi & Rossi (2003, p.1244) asks: "[h]ow is it possible to align the incentives of several different individuals"?

It is this selfsame problem the various efforts to rewrite PackMan is facing: how to align the interests of the community at large in order to mobilize the resources for rewriting? In the first attempt at rewriting PackMan, turning the package manager into "a true community project" goes through the four developers who will rewrite PackMan with a core system and "a solid API for components [where] major parts that are now core PackMan are going to be implemented as components". By framing a set of problems and actors whose interests are blocked by these problems, the four developers tries to mobilize resources (Callon 1986) for rewrite and replace PackMan. These translations are summarized in Figure 2 below.

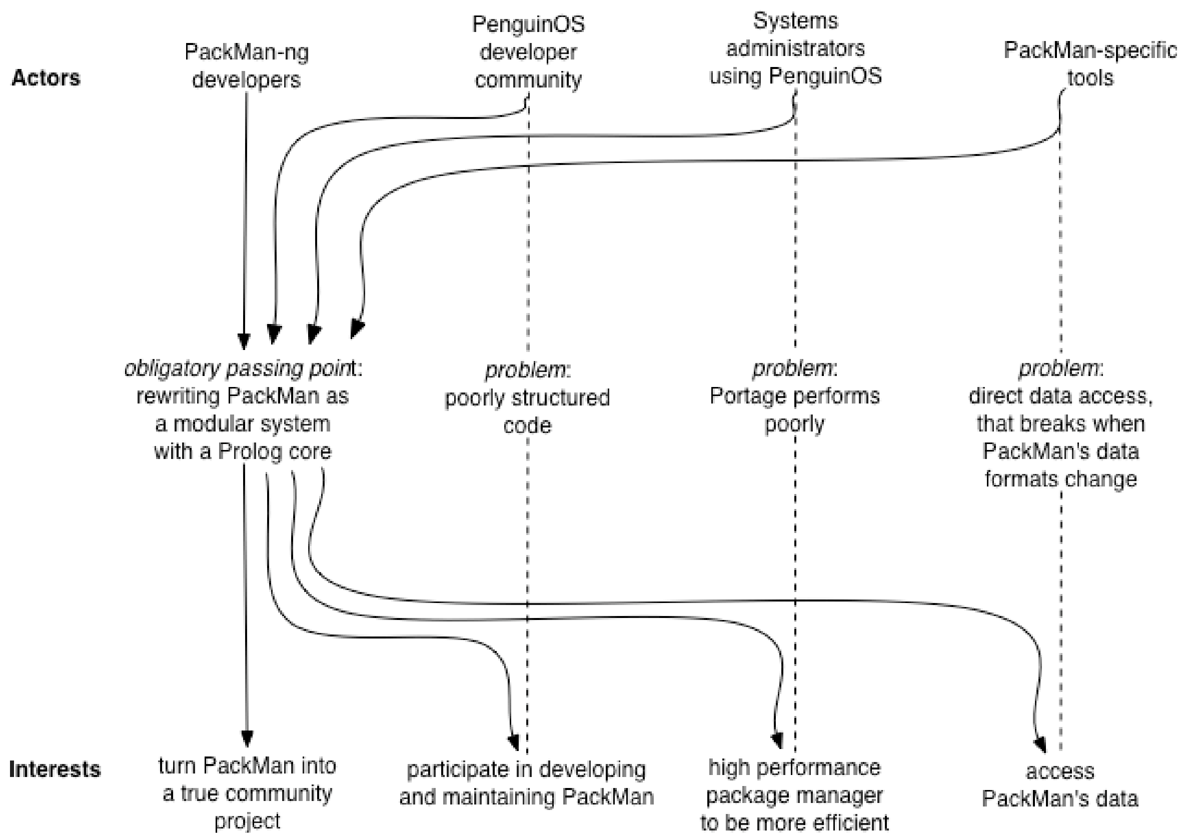

Figure 2 The PackMan-ng developers' translations

However, it is not only a question of mobilizing any odd resources. The problem of the next generation PackMan developers is that they want to mobilize particular resources. By translating interests into modules that clearly delineated boundaries between actors and their interests, and by inscribing these as boxes in an architecture diagram, the four developers make the architecture diagram stand in for their translations, making them more durable. Through the use of boxes, labels, and clearly separating between boxes, the architecture diagram provides an overview of dependencies between various parts of the architecture; in other words: it inscribes a sequence of work.

By saying that the programming language for realizing next generation PackMan is to emerge from the requirements, they are mobilizing resources to do the requirements work first, while leaving to the small next generation PackMan team to write the core system first. As such, the resources they want to mobilize are for writing the plugins. However, the effect of proposing Prolog in the design and for the prototype is that resources are spent in discussing implementation language details and problems with using Prolog. While the Prolog prototype is intended to act as a focal point for mobilizing resources for developing plugins, as it fails to materialize there is no mobilization and resources become scarce.

However, the explanation that resources is the reason why the rewrites failed has to been seen in as deeply embedded in and emerging from the context. It is

worth noting that although a number of objections over the plan for the first attempt at rewriting PackMan, nobody questioned the feasibility of the effort. Yet, six months down the line, the PenguinOS developers argue that lacking resources is why the effort failed. What has happened?

Resources are scarce because there is a competition for resources within PenguinOS, as well as the constant need to attract new developer resources. The whole PenguinOS effort relies on the sustained interest of users and developers. As observed with many large OSS projects, the key process for quality assurance is users reporting failures to the developers (Feller & Fitzgerald 2002). As Mockus et al. (2004) observes: the number of people reporting software failures greatly exceeds the number of developers. The sustained interest of user is therefore important for the PenguinOS community.

The mechanism for sustaining this interest lies in the continued improvement and enhancement of the software, "improvements and simplifications" as put in the above quote. What we see throughout the period is therefore that the existing PackMan application continues to change. Attracting new developers is a concern for the community, as the number of unresolved failure reports is continuously growing for PenguinOS. Adding functionality to PackMan is also seen as a way of recruiting new developers. A concrete example is the way Bob is recruited to the community by the promise that he can implement web application support for PenguinOS. However, being a member of the community involves responsibilities, and resolving failure reports is one of these responsibilities. So, recruiting new developers by adding new features to PackMan is not only a way of enhancing the software, but also a way of mobilizing resources for addressing the growing number of failure reports.

When the PenguinOS developer above questions how the PenguinOS community can manage without a PackMan update for 6 to 12 months, he is alluding to constant need for balancing between the need for technical stability for rewriting PackMan on one hand, and the need for adding new functionality to attract new development resources and keep existing developers interested in the project.

Transition strategy as a processes

Whereas in Jørgensen's (2001) description of the process of rewriting the FreeBSD kernel the scope of the changes and the sequence of actions seem unproblematic, we see that rewriting and replacing PackMan is a continuous process of negotiating over the scope of the changes to be made, their sequence, and which actors to be involved in the process. It is about formulating a transition strategy (Monteiro 1998) for the transition from one version of the package manager to the other.

Formulating this transition strategy is a process of continuously balancing numerous interests. On the one hand there is the interest in keeping stable the features of the software to be rewritten. On the other hand, use of the software to be rewritten continues to evolve and users have interest in the existing software to evolve accordingly. A balance must be struck between these interests. However, this balance point is continuously negotiated and renegotiated, and any attempt to rewrite the software has to remain flexible to these changes.

As much as formulating a transition strategy is about imposing stability of the entire package manager, it is a negotiation over what parts to keep stable and what to change. We see this in the focus in the attempts to rewrite PackMan: going from a complete rewrite of the whole artefact, to a modularization of the existing code, to the introduction of an API on top of the existing code. It is a longitudinal process of translation spanning months, during which the identity of actors and the boundaries of what is to remain stable with PackMan and what can change are continuously negotiated. The actors' margins of manoeuvre, their possibilities of making incontestable statements about the efforts to rewrite and replace, is delimited through this process of translation.

When one of the PackMan developers in hindsight says that rewriting PackMan from scratch "is a MAJOR waste of extremely limited resources", the statement tells us nothing about why next generation PackMan failed. Nor does it tell us anything general that rewriting software from scratch requires a lot of resources. Rather, the statement bears testament of how the PenguinOS developers' margins of manoeuvre is limited by the installed base. There is no longer room to state that it is possible to rewrite PackMan from scratch. Again, this does not provide us with the means to make generalized statements that rewriting software artefacts from scratch is never feasible because of a continuously changing installed base.

Furthermore, what we see is that to better control the process of rewriting and replacing, the boundaries of the involved actors are limited. From encompassing the entire PenguinOS developer community with the rewrite of next generation PackMan, the scope of involved actors are seriously reduced in both PackMan modularized and the attempts at writing an API on top of the existing code. When a PenguinOS developer in hindsight explains that "waiting for the community to provide requirements … doesn't work", the statement tells us nothing about why next generation PackMan failed. Nor does it leave us any margins of manoeuvre to make generalized statements about the number of actors involved that can be involved in successfully rewriting and replacing information systems. Rather, what it does tell us is that how the inertia of the installed base limits the PenguinOS developers' margins of manoeuvre in making statements about the number of involved actors in the process of rewriting and replacing software.

What we can generalize, however, is this. The formulation of a transition strategy is constituted through a continuous negotiation with the installed base.

This process of negotiation is a process of balancing the interests of the involved actors – both technical and non-technical. It is a process initiated by the construction of problems and actors with interests, but it is also a process from which new problems emerge. With new problems, existing actors change and new actors emerge. As interests "are what lie *in between* actors and their goals, thus creating the tension that will make actors select only what, in their own eyes, helps them reach these goals amongst many possibilities" (Latour 1987, pp. 109-110), new relationships between actors change. As actors and their interests change, so does that which lies in between them: the interests. As such, rather than being an end product in itself, the transition strategy is continuously formulated and reformulated through a process of continuously emergent problems, actors, and interests enables and constraints the task of rewriting and replacing PackMan.

Concluding remarks

In this paper we show how a transition strategy for rewriting and replacing OSS emerges from and continuously changes in response to the installed base. There is a mutual relationship between transition strategies and the context of use and development. The way transition strategies changes the context feeds back to change the transition strategy. We show how this mutual influence changes the transition strategy along three axes: the scope of the rewrite, the sequence to replace the package manager, and the actors to be involved in the change process.

While the entire PenguinOS community can agree upon the need to replace the existing system, we show how the existing system's ability to continuously meet the community's interests are greater than the perceived benefits of replacing the system. Although the introduction of an API on top of PackMan redirects existing connections to PackMan, the transition strategies of the PackMan developers were unable to redirect new connections to the existing PackMan code, like those made for web application support. We show that battling the inertia of the installed base, then, is not only about changing existing connections from the software being replaced towards its replacement (Hanseth and Monteiro 2002). It is also about the ability to redirect new connections to the installed base to the replacement software throughout the process of rewriting and replacement.

In order to understand and analyse processes of rewriting and replacement, it is therefore important to understand the rationalities and logics in play by different actors. It is important not only to take the actors' own explanations of the world for real, but also to understand the logic and rationality of their explanations in the eyes of the other actors without giving any undue privilege to either view. Furthermore, statements of the world need to be contextualized, when were they made and in response to what, in order for the information systems researcher not to be locked into single actors' views as true and thereby seeing other actors'

views as false. As information systems researchers it is also important not to lock on to and give priority to some actors' techno-economic rationalities, but rather to remain sensitive to our own academic techno-economic bias and challenge this through careful analysis of the statements made by those we study.

References

Bass, L., Clements, P., and Kazman, R. (2003). Software Architecture in Practice, Addison-Wesley, Boston, Massachusetts.

Bianchi, A., Caivano, D., Marengo, V., and Visaggio, G. (2003). Iterative reengineering of legacy systems. IEEE Transactions on Software Engineering, 29(3), 225- 241.

Bonaccorsi, A. and Rossi, C. (2003). Why open source software can succeed. Research Policy, 32(7), 1243-1258.

Callon, M. (1986). Some elements of a sociology of translation: Domestication of the scallops and fishermen of St. Brieuc Bay. In The Science Study Reader (Biagioli, M. Ed.), Routledge, New York, New York.

Crowston, K. and Howison, J. (2005). The social structure of free and open source software development. First Monday 10(2).

DistroWatch (2006). The top ten distributions: A beginners' guide to choosing a (Linux) distribution, http://distrowatch.com/dwres.php?resource=major. Last visited: November 25 2006.

Feller, J. and Fitzgerald, B. (2002). Understanding Open Source Software Development. Addison-Wesley, London.

German, D. (2005). Software engineering practices in the GNOME project. In Perspectives on Free and Open Source Software (Feller, J., Fitzgerald, B., Hissam, S.A., and Lakhani, K.R. Eds.), p. 211, The MIT Press, Cambridge, Massachusetts.

Glass, R. (1999). Of Open Source, Linux and Hype. IEEE Software, 16(1), 126-128.

Hanseth, O. and Monteiro, E. (1998). Understanding Information Infrastructures. Unpublished manuscript, available at http://heim.ifi.uio.no/~oleha/Publications/bok.html.

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

Klein, H.K. and Myers, M.D. (1999). A set of principles for conducting and evaluating interpretive field studies in information systems. MIS Quarterly, 23(1), 67-93.

Latour, B. (1987). Science in Action. Harvard University Press, Cambridge, Massachusetts.

Mockus, A., Fielding, R.T., and Herbselb, J.D. (2002). Two case studies of open source software development: Apache and Mozilla. Transaction son Software Engineering and Methodology, 11(3), 309-346.

Monteiro, E. (1998). Scaling information infrastructure: The case of the next generation IP in Internet. The Information Society . 14(3), 229-245.

Paulson, J.W., Succi, G, and Eberlein, A. (2004). An empirical study of open-source and closed-source software products. IEEE Transactions on Software Engineering, 30(4), 246-256.

Samoladas, I, Stamelos, I, Angelis, L., and Oikonomou, A. (2004). Open source software development should strive for event greater code maintainability, Communications of the ACM, 47(10), 83-87.