OSS Product Family Engineering

Jilles van Gurp
Nokia Research Center, Software and Application Technology Lab
jilles.vangurp AT nokia.com

Abstract

Open source projects have a characteristic set of development practices that is, in many cases, very different from the way many Software Product Families are developed. Yet the problems these practices are tailored for are very similar. This paper examines what these practices are and how they might be integrated into Software Product Family development.

1. Introduction

The notion of software reuse has been studied and practiced for decades. Over time, the attention in the technological dimension has shifted from subroutines to modules, frameworks and finally Software Product Families. In the organizational domain, focus has grown from code reuse by the author of the code to code reuse by others than the author of the code working on the same software, working in the same organization and finally between organizations. Software Product Family engineering is very much about intra-organizational reuse.

The open source movement was born out of a pragmatic need to share code among individuals. This need arose in the late sixties and early seventies when researchers started to share code for common assets such as compilers, system libraries and later operating systems such as UNIX. During the eighties, the practice of code sharing was given a legal framework in the form of license agreements such as the BSD license and the GNU public license. Finally, during the late nineties, when Linux emerged as a mainstream operating system, the term open source started to be used to refer to this practice of collaborative development, licensing and distribution of software.

Currently a wide variety of programs, components and frameworks is available under an open source license. Many software companies now depend on open source components for their core business. For example, the Gnu Compiler is widely used across the industry and crucial for many embedded system companies. Similarly, the Linux operating system kernel is used by many embedded systems companies. Even Microsoft is known to use BSD licensed components in e.g. their network stack.

Open source components form a rapidly growing, shared repository from which, depending on the specific license, anybody can just take what they need and use it. Open source is very much about inter-organizational reuse.

It turns out that, as the scale of development is growing, inter-organizational reuse is increasingly important. Few organizations can afford to develop everything in house. For some years, COTS have been pushed as the solution for this problem. However, lack of source code, support, perfectly matching feature sets, and other factors have prevented the widespread adoption of COTS.

However, many organizations are now replacing their non-diversifying, in house developed components with open source, or even making their entire software available as open source (e.g. Sun). Open source is succeeding where COTS has failed.

Open source software is enabling interested parties to share code under a legal umbrella that sufficiently protects the rights of the using and producing parties. The use and production of OSS in the context of Software Product Families is both an obvious and inevitable solution to the problem that in house developed software is an increasingly smaller (relative, not absolute) portion of the total amount of software required. Eliminating non value adding development in Software Product Family development is key to reducing cost.

Arguably, open source development and Software Product Family development can claim to represent the two most successful strategies for reusing software. This position paper explores several of the practices common in open source communities with examples from three major open source projects (Eclipse, Mozilla, and Linux). Additionally some discussion is presented on how these practices may be applied to Software Product Family development.

1.1 Remainder of this paper

The rest of this paper consists of three parts. First (section 2) we characterize more precisely what we

understand the OSS development practice to be. Then we illustrate this with three open source projects: Eclipse, Mozilla, and Linux. Finally, we reflect in section 4 on how the identified practices could be integrated into Software Product Family development and we conclude our paper in section 5.

2. OSS development practice

Open source in the narrow definition refers only to the license used to make the software available. As such, the use of open source is completely orthogonal to the use of Software Product Family development practices (i.e. one could develop a Software Product Family using the conventional methods for doing so and then make the resulting software available as open source). However, in its wider definition it may also be understood to include a set of development practices and a certain style of development that is very different from the way Software Product Families are developed by many organizations. In this section, several of these practices are discussed.

2.1 Communication

Many open source projects are developed by people that are geographically distributed, may be in different time-zones and work for different organizations. Consequently, many forms of communication that are common in enterprises such as phone calls, face to face meetings are impractical. Additionally, the practice of one individual (a.k.a. the boss) telling other individuals what to do is not that common. Decisions are based primarily on consensus rather than authority.

In the open source community, email and IRC are the primary means of communication rather than face to face meetings. Technical discussions are preferably conducted on mailing lists which are generally archived for future reference. IRC or similar instant messaging tools are used for a more direct style of communication. These conversations tend to be less formal and they are generally not archived. Cases are known of OSS developers sending each other emails while sitting at the same table for the purpose of archiving the discussion or simply conducting it in public.

2.2 Tool centric development

A key characteristic of open source development is that open source projects are organized around a set of enabling tools. Generally, these tools (in addition to the usual development tools such as compilers and IDE's) include:

- A version management system. CVS is historically popular in open source projects but is now rapidly being replaced by the much more modern Subversion (e.g. the Apache Foundation and Sourceforge use Subversion nowadays).
- A bug tracking system. Bug tracking systems are commonly used both for tracking bugs, requirements and even project planning. Many open source projects require any change committed to the version management system to be related to a bug or issue in the bug tracking system.
- WIKI's are increasingly popular for document management. Particularly end user documentation, development documentation and project documentation (e.g. roadmaps) tend to be maintained in WIKI's.
- Build and integration tools (e.g. maven, ant, Make). Many open source projects depend on automated builds, integration and testing tools for receiving feedback about project progress and status.

Open source development is necessarily tool centric because its developers are generally distributed geographically. The tools are effectively their only interface to the project. Consequently, development practices that are incompatible with this interface are rarely found in open source projects. It therefore is quite common for OSS projects to not have explicit design documentation; use case diagrams or even an architecture design phase. However, that does not mean that such projects do not have architecture, design and requirements.

Instead, these assets, insofar deemed relevant by the developers, exist in the tools. Use cases are rare but detailed requirements and requirements change requests are managed through the bug tracking system. Architecture documentation is generally lacking but then the audience for such documentation is not necessarily the developers either in organizations that do write architecture documentation.

2.3 Strong code ownership

Though the source code of an OSS project may (legally) be modified and redistributed by anyone the actual occurrence of someone taking open source software, modifying it and distributing it independently from he original (a practice known as forking) is quite rare. Generally, open source projects have strong ownership with a small group of developers coordinating and guarding the development.

Source code ownership is governed through version repository access rights. Typically, a limited set of individuals has the right to make changes to particular directories in the version management system. It is also quite common that approval of key individuals is needed to make any kind of change. The strong ownership enforces code reviews take place and that changes are tested properly.

2.4 Technical roadmap

Unlike commercial software development where managers, customers and other stakeholders determine what is developed, the evolution of open source software projects is primarily determined by:

- Developer interest. Developers generally prioritize features that they are personally interested in.
- Corporate funding. Most large open source projects are developed by developers who are paid to work on the project. Of course, the reason they are paid is that their companies have a strategic interest in the project and presumably want to influence the direction of the project.
- Project organization. Many open source projects are led by a small group of, more or less, independently operating individuals whose personal vision strongly influences technical decisions made in the project.

In order to prioritize features or make major technical changes to the software, interested parties need to work in this structure. They need to convince whatever individual is in charge that the suggested change is a good one; generate interest among developers to actually get the change implemented and maybe arrange some funding to allow developers to work on the change.

2.5 Quality management

A consequence of developers being in charge of the technical roadmap is that generally developers prioritize quality attributes that interest them. For example, the open BSD project has a strong security focus. The open BSD lead developers all have strong engineering backgrounds in security related matters. Its products are generally considered to be of exceptional quality in this regard (e.g. open SSH or the open BSD kernel). Additionally, any issues related security are handled swiftly once the developers are notified of them. Other quality issues outside the scope of the developer's interest receive much less attention (for example, usability is often sacrificed in favor of configurability).

Similar to the technical roadmap, the quality management can be influenced through funding, argumentation, etc.

2.6 Release Management

Release management is the process of converting source code in the version management into a stable, well tested software package that can be distributed to end users. Many open source projects have well defined processes for producing a release. Generally, there are a few differences with comparable processes in commercial projects:

- The software is released when it is 'done'. This moment is generally agreed on either by leading individuals in the process or by consensus. Despite this, many open source projects try to follow date driven roadmaps where milestones and releases are planned to occur. In commercial projects, such deadlines tend to be much harder and inflexible, however.
- The software release is preceded by a series of public alpha, beta and release candidate milestones. During this period, interested third parties not taking part in the development test the software and provide feedback. Though technically it is possible for them to use so-called nightly builds straight from the version management repository, few people outside the developer community are actually willing to take the risk.
- Because the eventual release is scrutinized in public, quality tends to be high (in so far of interest to the involved users and developers).

Especially for large open source projects, the release process tends to be well defined.

3. Examples

To illustrate the claims made in the previous section, we present three case studies which highlight all of the practices mentioned in three large open source projects with solid reputations in the software industry.

3.1 Eclipse

The Eclipse foundation is responsible for the development of the Eclipse IDE and a rapidly growing number of associated software packages (plugins). Originally, the Eclipse source code was contributed by IBM who still provides a significant amount of funding. However, the Eclipse foundation is now an independent organization that oversees the development. In addition, other companies, including competitors of IBM, now contribute funding and development resources to the foundation.

Communication. Communication happens primarily through email, IRC, the Bugzilla bug tracking system, the WIKI website, mailing lists and the Eclipse.org website. Eclipse developers are distributed across the globe and mostly employed by (competing) corporations (e.g. BEA and IBM).

Tooling. Eclipse source code is maintained in a CVS repository, Bugzilla is used as the bug tracking system and project documentation is divided between the Eclipse.org website and the Eclipse WIKI. Additionally there are several mailing lists both for end users and developers.

Code ownership. The Eclipse foundation restricts write access to their code repository. Generally, the

process for contributors involves contacting a so-called committer for making a particular change. Typically, components have an owner and multiple committers. The role of the owner is to coordinate the work on that component. When receiving an external contribution, the committer either commits the change or (limited) commit rights are given to the new contributor [2]. A key element in the process is assuring that the contribution conforms to the legal framework which involves topics as copyrights, the license, patents and export rules concerning cryptography technology [1]. All contributions must be traceable and accountable. Procedures like this are common to many open source projects.

Technical roadmap. The Eclipse project strongly depends on development resources contributed by various software companies. Those companies have a strong influence on what is developed. A good example is the web tools project, a massive undertaking by IBM, BEA and several other companies to create a set of J2EE development plugins for the Eclipse IDE. Over the course of 1.5 year, this project went through a set of planned milestones with specified sets of features and managed to release a feature complete 0.7 release for the Eclipse IDE 3.1 release, a more mature 1.0 release half a year later and recently a 1.5 release. The input for the project was a set of contributed development tools from various vendors and a number of (public) J2EE specifications that these companies wanted to have support for.

Quality management. The core Eclipse project has seen many changes related to improving performance and memory usage in its recent versions. To accomplish this, the automated test suites that are run on nightly builds of the Eclipse software have been extended with tests to measure specific scenarios. Furthermore, target performance numbers have been defined and cases where performance targets are not met are treated as bugs. The test reports for the nightly builds and release candidates of the Eclipse 3.2 release list performance numbers relative to the 3.1 release. Each case where performance decreases is treated as a regression. Aside from performance, the nightly builds also include a large number of unit tests (thousands). Specific quality issues either identified automatically or through testing, are reported in the bug tracking tool.

Release management. The Eclipse project has well defined release cycles which are beyond the scope of this article to discuss in full. The key philosophy of the Eclipse release process is to be automation centric. The release practice is outlined in a FAQ [3] that provides answers on mostly technical topics such as how to set up the test suite; how to integrate components into the build process, etc. Effectively, the build infrastructure implements and enforces a sophisticated system of checks and balances that ensures that produced releases meet predefined criteria.

In addition to the technical constraints, the release process is complemented by communication and coordination from project leads through the mailing list on such topics as roadmaps, schedules, code freezes, test plans, etc.

3.2 Mozilla

The Mozilla foundation which oversees the development of Firefox browser, the Thunderbird mail client and a number of related software projects has a similar history to the Eclipse foundation. Originally, the Mozilla browser was contributed by Netscape. The company Netscape has since been absorbed into AOL and was eventually liquidated. During this process, the Mozilla foundation was created which still employs some former Netscape employees but also a growing number of new employees. Similar to Eclipse, the Mozilla foundation receives corporate funding from a number of companies that have an interest in the continued existence of the Mozilla technology.

Communication. Similar to the Eclipse developers, the Mozilla developers are also distributed globally. In addition, they use similar communication tools.

Tooling. Similar to Eclipse, Mozilla development is very tool centric. In addition, Mozilla is famous for inventing its own tools. For example, Bugzilla is one of the software projects that is maintained by the Mozilla foundation. Other tools created by Mozilla include Bonsai for examining the CVS history, LXR for browsing the cross referenced source code through a web site, Tinderbox for monitoring the build process and Litmus for managing and running automated tests on Firefox. Many of these tools, most notably Bonsai and Bugzilla, have been adopted by other projects and have even been integrated into commercial tools.

Code ownership. The Mozilla project features strong code ownership. In practice, this means that every patch must be reviewed and approved by a component owner before being committed [4]. Component owners are generally either Mozilla foundation employees or individuals with a long history in the project employed by one of the high profile donating corporations (e.g. The Firefox project leader Ben Goodger is a Google employee).

Technical roadmap. Firefox development takes place in the context of a roadmap which is updated at regular intervals (once or twice per year). The roadmap features milestones and releases with a list of features and corresponding Bugzilla ids. While the foundation strives to release according to the roadmap, the Mozilla

release policy in practice appears to be much less rigid then e.g. the Eclipse project. Often releases are delayed for weeks or even months (as long as is needed). Also new milestones may be inserted into the roadmap. Finally, the roadmap acts mostly as a guide rather than a complete functional specification. It contains what the project leaders believe are relevant features to work on. Input for this comes from the mailing lists, the WIKI and IRC discussions.

Quality management. The Mozilla project has a number of quality attributes that are explicitly managed:

- Code quality. As part of the commit process, each patch is attached to a bugreport in Bugzilla that describes the problem and solution(s). Before being committed, the patch is reviewed and super reviewed.
- Correctness. The Firefox browser implements a large number of open standards. In addition to that it supports poorly defined incorrect interpretations of these standards (a.k.a. the quirks mode) of other browsers. Testing for compliance therefore is an extremely complicated affair that is supported by manual testing, half automated tests (a.k.a. smoke tests) and fully automated tests (e.g. using the Litmus tool).
- Performance. Similar to correctness, performance is explicitly managed through testing (automated and manually).
- Security. Browser security is of extreme importance to end users. In addition, it is a sensitive topic. Therefore, the Mozilla project has well defined procedures for reporting, solving and publicizing security issues. Additionally, recent versions of the Firefox browser include an auto update feature to stimulate rapid deployment of security related bug fixes.

Release management. The Mozilla foundation manages and oversees the release process. Generally the process involves a number of alpha release milestones followed by more or less feature complete beta releases (typically two) and finally followed by a series of release candidates (as many as is needed). During this process, the rules for committing changes become stronger. During the release process, no changes are committed before being extensively discussed by project leads. Additionally each of the milestone and beta releases has a mini release process which involves a few days of testing candidate builds and restricting commit access to the CVS repository.

3.3 Linux

The Linux kernel development is overseen by its inventor Linus Torvalds. The style in which he manages the project is very different from Mozilla and Eclipse though still tool centric. Unlike the former two projects, Linux development is traditionally much more fragmented among thousands of developers and hundreds of contributing companies. In a recent interview, Torvalds estimates that there are around 50 developers he communicates with directly and he estimates that through them he is in contact with approximately 5000 kernel developers [5].

Communication. Linux kernel developers rely very much on mailing lists and private mail exchanges (or IRC conversations). Linus Torvalds style of leadership has often been referred to as that of a benevolent dictator: ultimately, he is the one who takes important decisions though in practice this responsibility is delegated to trusted individuals.

Tooling. The central leadership is also reflected in how the tooling works. The Linux project recently switched from using Bitkeeper to its own developed tool Git. Both are so-called distributed version management tools. Rather than pushing changes to a central repository, the lead kernel developers pull changes into their private repositories either by accepting patches from a mailinglist or by updating from somebody else's repository. The repositories available at kernel.org are read only for most developers. They are merely the places where lead developers publish their approved change sets from their private repositories. Other tools used in Linux development include Bugzilla and various news groups. However, email remains the most important tool.

The use of a distributed version management system on a large scale is a recent innovation that no doubt will be followed up by adoption in other projects as well. It has proven to be an effective way to orchestrate the development on a large software system with many active developers.

Code ownership. As the central leadership suggests, code ownership is very strong in the Linux project. To get a change committed in the Linux kernel the associated patch needs to be communicated by email to the relevant people that have the right to approve the change. Eventually the change will find its way to Linus Torvalds, who, after assuring that everything has been properly reviewed, approved and tested may or may not include the change at his discretion.

Technical roadmap. Linux development tends to be more anarchistic than Mozilla or Eclipse development. Essentially, there is no centrally maintained roadmap. Development consists of many subgroups working on e.g. drivers, new memory management routines, etc. Major versions of the kernel usually include some re-architecting as well. E.g., the current 2.6 version

included features to allow the kernel to scale better on distributed systems.

Quality management. Stability, performance, security, modularity are all important themes in the development of the Linux kernel. Linux is used on many mission critical servers, mainframes and desktops. Additionally it is embedded in devices. Therefore, all these quality attributes are critical. Despite this, there are few quality management tools or processes in the Linux development. Code review and testing by users seems to be the main way of controlling quality. The reason for this is that the Linux development and user community is extremely diverse. There are thousands of developers working on or depending on the latest kernel sources. Testing happens in a distributed fashion on a wide variety of devices by a wide variety of parties with a wide variety of interests (device drivers, processor architectures, file system development, real time behavior, ...). The testers include: individual desktop users, hardware vendors, Linux distribution vendors, and the developers themselves.

Release management. In principle, Linus Torvalds is the one who declares a release. His principle over the years has always been that "it's done when it's done and not sooner". Despite this, the process seems to involve a number of stages spanning several months during which progressively less changes are accepted and testing efforts are increased.

4. Improving SPF development practice

Open source development as outlined above represent the state of the art in the way software developers believe software should be developed. If left to their own devices, this is how they self organize.

In many respects that is very similar to how development takes place (or should take place) in traditional closed source environments. However, there are some differences. In this section, we examine how the practices discussed above may be integrated into Software Product Family development practice.

4.1 Communication

Software Product Family developers are faced with similar communication challenges as open source developers. Often development teams are large, may be geographically distributed and composed of different organizational entities. Additionally, a growing need for accountability (e.g. for legal reasons) makes it obvious that the solution to this communication challenge also needs to be similar (see e.g. [1] for the process for accepting contributions in the Eclipse project).

Additionally, many multinational companies are so large that the challenge of getting their developers to work together on projects requires a more or less similar communication infrastructure to the OSS style of communicating. Email remains an important tool across such organizations. Consequently, many of the open source communication tools are already finding their way into the corporate world (e.g. WIKI's, bug tracking tools and instant messaging tools).

A problem remains that, in general, only the developing part of such companies uses such tools. Senior managers, sales departments and other parts of the organization are not using the same tools for communicating. This creates a conceptual gap between the development reality on the work floor and the management reality. The alternate management reality is an appealing ground to make important decisions that have major effect on the development reality: especially for people who should not be making those decisions.

The term slideware refers to software entities that only exist in PowerPoint slides and not in the relevant development tools [6]. The problem with slideware is that it doesn't have any corresponding representation in the development communication infrastructure. Once it does, it ceases to be slideware. Until it does, it does not exist. Problems arise when slideware fails to materialize in a timely fashion.

In open source projects, slideware does not exist. New requirements for features become WIKI documents. WIKI documents become bug reports. Bug reports are commented on and eventually are closed with either a reference to a patch or CVS commit or a message as to why the particular feature is no longer relevant.

4.2 Tooling

In a corporate setting tooling tends to be better (e.g. the use of commercial version management or document management tools is common; additionally expensive modeling tools, IDE's and other tools may be used). However, over the years, the open source community has produced its own set of tools that meets its requirements. Such tools include everything from the, now, industry standard GCC compiler, Bugzilla, the Mozilla tool chain outlined above to sophisticated distributed version management systems (e.g. Subversion and GIT). Many commercial development tools are simply based on open source components and either add value through support or by adding specific features.

A key feature of tools in the open source development community is that they are developer centric. Their primary objective is to make the developer's work (i.e. developing software) easier. Many tools used in Software Product Family development on the other hand are not developer centric (or even developer friendly). For example, many variability management

tools are aimed at requirements engineers or even sales departments; many architecture modeling tools are used by senior architects to communicate to their managers; UML modeling tools are used to document already developed software; model driven architecture tools are aimed at the consumers of the software (i.e. the people that design products) rather than the developers of the composed software. Often bureaucracy in the form of heavy processes is needed to enforce the proper use of such tools.

A key lesson that may be drawn from the open source style of tooling is that in order to be effective, tools need to integrate into other tools. The set of tools create their own reality in which the developer is active. Anything outside this reality integrates poorly into the communication structure used and quickly becomes irrelevant (for the developers). OSS developers seem to have little or no need for such tools and yet manage to scale development to impressive levels of scale, speed and quality.

A good example of an integrated tool from the OSS community is Bugzilla. In both the Mozilla and Eclipse projects (and in many other places) this tool is not only used for bug tracking but also for requirements engineering, release management and even process improvement. The imposed reality in these projects is that any change to anything is communicated through and documented in Bugzilla. Bugzilla in turn is integrated with email (notifications) and version management systems.

Many Software Product Family tools are plagued by a lack of integration. Design documentation tends to be incomplete (or non existent) because the document management system is not part of the development environment, variability management tools depend on extensive manual updates to stay in sync with source code level changes; requirement specifications need to be continuously validated and verified. Successful examples do exist however. For example, KOALA, the architecture description language used by Philips integrates with the build system and design [7]. The COVAMOF variability management tool proposed by Sinnema et al. integrates into visual studio [8].

A second problem with such tools is that they are not general purpose. This poses problems when product families become product populations and different sets of incompatible tools become obstacles that need to be bridged. A key driver for growth in the OSS communities is that everybody uses the same or similar tools. This lowers the barrier of entry for new contributors. The fact that the tools are comparatively primitive is compensated by the fact that everybody knows how to work with them. Similar consolidation in SPF development tools is required as SPF are increasingly complemented with third party provided software components (open source and closed source).

4.3 Code ownership

While corporate interest in many OSS projects is huge (also financially), OSS projects tend to be self organizing in the sense that all important decisions are made by developers rather than managers. The relevance of opinions of individual developers is strongly related to their level of (technical) contribution to the project (within the Eclipse project this is called a meritocracy).

A key issue in Software Product Family developing companies, which are generally not organized as meritocracies, is that decisions are made based on authority, rank and status in the company. Especially when difficult technical decisions are taken, this may not be the most optimal strategy since it is not common that the person with the most authority also has the most technical competence. At best, he or she has the wit to trust the judgment of the competent subordinates who should be making the decision. In other words, important technical decisions are routinely taken by the wrong people; influenced by the wrong motives (e.g. short term market interests vs. quality) and misguided by a lack of relevant knowledge of domain, technology and software design.

To counter this problem, many organizations organize their Software Product Family development as a separate organizational entity to shield it from the short term interests that are present in depending organizational units that develop the products [9]. Despite this, influence of the other organizational units remains high through e.g. funding, upper management etc.

The conflict between the long term technical roadmap (development), the short term market interests (sales) and the long term market perspective (marketing) poses a risk to the long term technical health of the software. Open source projects solve this by being autonomous. That does not mean they are not affected by the market. Through funding, donations and man power companies exert influence over the technical roadmap, short term interests etc. For example, IBM maintains a strong influence in the Eclipse project (and in fact many other open source projects that are of strategic interest to them). While they cannot dictate their changes, they have a very strong influence on the technical direction of their project simply by funding development of features and components that are of interest to them.

4.4 Technical Roadmap

Software Product Families are a key investment for the companies that own them. Naturally, these companies

wish to have a strong influence on the roadmap of their product lines. As outlined above under code ownership, this can easily lead to a situation where decisions are made by the wrong people. A real problem is that these roadmaps tend to focus on functional requirements only (because that is what is marketable to customers).

For example refactoring is unlikely to feature in a SPF roadmap. Yet, when looking at OSS projects, refactoring is often a driving force for major new releases. For example, the Eclipse project was refactored extensively between version 2 and 3. In addition, the subsequent 3.1 and upcoming 3.2 have seen additional refactoring work done. This has lead to major improvements in performance, usability and flexibility (which was the main reason for the refactoring). Additionally, it has enabled the development of new features. The Linux kernel has seen large portions of its code being rewritten several times in its 1.0, 1.2, 2.0, 2.2, 2.4 and 2.6 incarnations. Firefox started out as an attempt by a small group of individual Mozilla developers to refactor/rewrite the Mozilla user interface, against the explicit wishes of their AOL peers at the time. Firefox has since replaced Mozilla as the flagship product of the Mozilla foundation.

Refactoring is a good example of an activity that developers will put on a roadmap and companies will likely not until the need becomes obvious. Refactoring almost always conflicts with commercial product roadmaps and short term interests of companies.

A problem with OSS roadmaps is that they reflect what the developers would like to see done, which is not necessarily as important for end users or relevant for the companies financing the development. Clearly, this model is not applicable to commercial software development on Software Product Families. On the other hand, there is a much better understanding of the technical feasibility of requirements at the developer level than there is elsewhere in an organization. An SPF roadmap should be realistic in the sense that its requirements are technically feasible, desirable and in the sense that important development activities needed for maintaining or improving quality are covered.

4.5 Quality Management

Open source development relies on three powerful quality management tools: large scale testing by end users, code reviews and automated tests. Testing on a large scale may be impractical for some software product families. But both other approaches are not unique to the open source community and can and should be implemented in software product family development methodology (in so far that is not the case already).

What make code reviews particularly effective in open source communities is that they can block the commit of a change until the component owner decides that the quality of the commit is good enough. This aspect of code reviews is hard to duplicate in companies where the code reviewer generally has limited authority to block changes (especially if they address urgent issues through a quick hack). Automated tests and test driven development are also increasingly popular. For example, in earlier research we reported on the successful use of automated tests in improving quality in Baan ERP. Test driven development is a cornerstone of extreme programming [10].

4.6 Release Management

Depending on the number of customers for a particular piece of software, the release process can become quite sophisticated. For example releasing a new version of the Mozilla Firefox browser is a process that spans multiple months and involves exposing alpha, beta and release candidate versions to large groups of users and processing any feedback that comes back from these users. In Software Product Family development, the number of users is typically small. Despite this, it may be productive to have some form of release process in place. It also depends on the organizational model. If, as outlined above, the product family development is developed by a more or less independent organizational entity, it makes sense that the rest of the organization does not access the version repository directly and instead relies on properly packaged and tested releases provided by the product family developers. However, having no feedback from real users (i.e. the product developers) until after the release is likely to cause issues with respect to implemented requirements and faults that are discovered after the release.

The author's experience as the (ex) release manager of a Dutch content management Software Product Family suggests that a good strategy may be to expose increasingly large groups of internal developers to increasingly mature versions of the product. Combined with a transition period with e.g. bi weekly releases this ensures that feedback and development stability (for the product developers) are balanced. This is similar to the beta stage of many open source projects where typically third parties (at their own risk) get involved into testing the beta and release candidate releases.

5. Conclusion

This position paper looks at open source development practice and makes some observations as to how this practice is different from Software Product Family development practice and how improvements could be made to the latter.

This article does not, and cannot possibly tell Software Product Family owners how to develop their software. Instead, it merely suggests to them that there is this set of practices that may be found in many open source projects that is known to work well at least in that context. In so far these practices are not already integrated into the Software Product Family development practice, it is further outlined how that might be accomplished and what the tradeoffs are.

The key vision underlying this paper is that from the point of view of the experts, i.e. the developers, the open source style of working is the best practice in the context of large software projects that are worked on by many geographically distributed developers.

A key difference between open source projects and most Software Product Families is that in open source projects the developers are in charge. This works out surprisingly well for all the aspects discussed above. All of the three cited projects are performing excellent in terms of quality, features and development speed. Therefore, the key recommendation of this paper to Software Product Family owners is to carefully (re)consider the balance between product family developers and management. Empowering developers allows them to work in a way that they consider best (and who are we to disagree). At the same time, of course the point of Software Product Families is directly aligned with the owning company's core business.

6. References

[1] Eclipse Foundation Process for accepting contributions, http://www.Eclipse.org/legal/EclipseLegalProcessPoster-v1.2.4.pdf

[2] The Eclipse development process. http://www.Eclipse.org/Eclipse/Eclipse-charter.html.

[3] Eclipse Release Engineering FAQ. http://WIKI.Eclipse.org/index.php/Platform-releng-faq.

[4] Hacking Mozilla, http://www.Mozilla.org/hacking/life-cycle.html

[5] Interview with Linus Torvalds http://edition.cnn.com/2006/BUSINESS/05/18/global.office.linustorvalds/

[6] Slideware definition, http://en.WIKIpedia.org/WIKI/Slideware, 2005-05-31

[7] R. van Ommering, Building product populations with software components, proceedings of the 24rd International Conference on Software Engineering, pp. 255-265, 2002.

[8] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, Jan Bosch: Modeling Dependencies in Product Families with COVAMOF. ECBS 2006: 299-307.

[9] Jan Bosch, Maturity and Evolution in Software Product Familys: Approaches, Artefacts and Organization. SPLC 2002: 257-271.

[10] Kent Beck, "Test Driven Development", Addison Wesley, 2002.