Towards a Pattern Language for FLOSS Development

Christoph Hannebauer Vincent Wolff-Marting Volker Gruhn

paluno - The Ruhr Institute of Software Technology
University of Duisburg Essen
Gerlingstrasse 16
45127 Essen, Germany
{christoph.hannebauer | vincent.wolff-marting | volker.gruhn}@paluno.de

ABSTRACT

There is a lot of research anticipating a "Free, Libre and Open Source Software" (FLOSS) development process and recurring characteristics of FLOSS projects have been discussed by various authors. Research suggests that a unique FLOSS development approach does not exist and there is a family of different development processes instead. Pattern Languages have been used to describe distinctive and common features of processes. In this paper, we identify four FLOSS development patterns derived from related work and discussion about FLOSS in the communities. Building on that, we propose methods to verify the patterns.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*productivity, programming teams*; K.6.3 [**Management of Computing and Information Systems**]: Software Management

1. INTRODUCTION

In his famous essay "The Cathedral and the Bazaar", Raymond was the first in describing a development style he associated with a "great babbling bazaar" [17], which became generally known as the "Open Source software development process". This term is misleading since Open Source software, in the meaning of software with publicly accessibly source code, can be developed using any kind of process. Raymond himself claimed that he "had already been involved in Unix and open-source development for ten years" using the cathedral-style development model. More recently, the more specific term "Free, Libre and Open Source Software" (FLOSS) has been used commonly in place of "Open Source". But the definitions [16, 7] of software, which is available with broad modification rights at no cost, still do not make any assumptions about the underlying development process [5]. Companies using any kind of development process can release their source code under a public license and thus qualify to use the term FLOSS.

PLoP '10, October 16-18, Reno, Nevada, USA.

In spite of these facts, numerous researches exist comparing closed source software to FLOSS [13, 19] or analyzing properties of *the* FLOSS development process, like quality of FLOSS projects [4] and the composition of FLOSS development teams [12]. However, the mean number of developers in FLOSS projects is one, which Krishnamurthy first showed for 100 mature FLOSS projects [12] and later Healy and Schussman showed it for all projects that Sourceforge had in its database in August of 2002 [10]. Healy and Schussman concluded that having only one developer in a software project contradicts the use of a bazaar development style as proclaimed by Raymond [17]. Another result of Healy and Schussman is that possibly quite different kinds of "mechanisms" exist in FLOSS projects, because different measures in FLOSS projects have consistent rank distributions and still "different kinds of activities may cluster in different sorts of projects" [10]. This empirical evidence against the usage of only one FLOSS development process in all software projects with a FLOSS license was later confirmed by Gacek and Arief, who have shown that some characteristics of FLOSS projects vary from project to project while some others are always the same [8]. With the rise of agile development methods in closed source development processes, the distinction between the closed source "cathedral-style" development and Open Source "bazaar-style" becomes more and more problematic. For example, Warsta and Abrahamsson compared Open Source development with agile methods and found them "rather close" [20].

Kelly has conjectured that Open Source could be a pattern language [11] and this paper attempts to build the foundation for a FLOSS development process pattern language. With the components of FLOSS development processes formally described in a pattern language, comparisons with other development processes can be more precise. The distinction between FLOSS development and closed source software development does not have to rely merely on the licensing model but on specifics of the processes themselves. It will also be possible to compare FLOSS development processes with each other. With a FLOSS pattern language, it is possible to extract valuable patterns used in FLOSS development processes and use them in other development processes. Further, parts of FLOSS processes that are problematic can be identified and improved.

Therefore, the target audience of this paper are researchers of FLOSS development processes and members of the FLOSS community seeking to improve their development processes. With the help of this pattern language, it is possible to grow a community around a project and keep it up, to increase

the number of features, and to improve software quality.

2. PATTERNS

When analyzing the structure of software development processes, we distinguish between *organizational patterns* and *project characteristics*. An organizational pattern as defined by Coplien and Harrison describes a possible solution to a local problem of conflicting forces [3]. A project characteristic is a property of the software project that influences or causes these forces but is not a behavior of the project members. For example, the number of developers or users that are involved in a project influences the development process and it can be determined which processes are appropriate solutions to the team's problems. Still, no specific behavior is required just because of the team's size. Therefore, project characteristics have a strong influence on the patterns to be used in a project and vice versa. This section will focus on organizational patterns but not on project characteristics. We will regard project characteristics in the statistical analysis that we will use to verify the patterns we have found.

We identified the patterns described in this section in scientific publications and discussions about FLOSS in communities. Guidelines and ideas for the realization of FLOSS projects have already been published [17, 8, 15, 6], but not in a structured way like patterns organized in pattern languages. Also, the most comprehensive publication of this kind, Raymond's essay "The Cathedral and the Bazaar", does not only contain guidelines each of which corresponds to a pattern but also common regularities in FLOSS processes for which a pattern does not necessarily exist.

Our intention is to specify a pattern language such that every FLOSS development process maps bijectively to a sequence of patterns. However, we have not yet verified the pattern language empirically and further research may show that additional patterns are required in a FLOSS pattern language that comprises widely used development processes of FLOSS projects.

This section contains the patterns we have found most important. Every pattern begins with an icon that symbolizes the core idea of the pattern. Then the pattern itself follows and eventually the last part depicts how researchers can verify usage of the pattern. This last part comprises a subsection Sources with evidence in literature for the pattern described and a subsection verification with criteria to empirically determine whether a specific FLOSS project uses the pattern.

Verifying the patterns on a successful FLOSS project yields a set of patterns that have proven to be usable for this project. It is possible that the project has used less patterns before and that it will use additional patterns in the future. Given only one snapshot of the patterns used by one project, it is not possible to see the complete order in which the patterns were applied. However, if a pattern A depends on a pattern B, a project will not use pattern A without pattern B at any point in time. By evaluating the set of used patterns in multiple successful FLOSS projects, those dependencies can be tested and possible sequences will become visible. This makes it possible to evaluate the relations between the patterns as they are used in practise. As a foundation for further empirical research, Figure 1 shows a pattern map that has not yet been verified with actual FLOSS projects. Patterns marked with (§) are not yet described or described elsewhere.

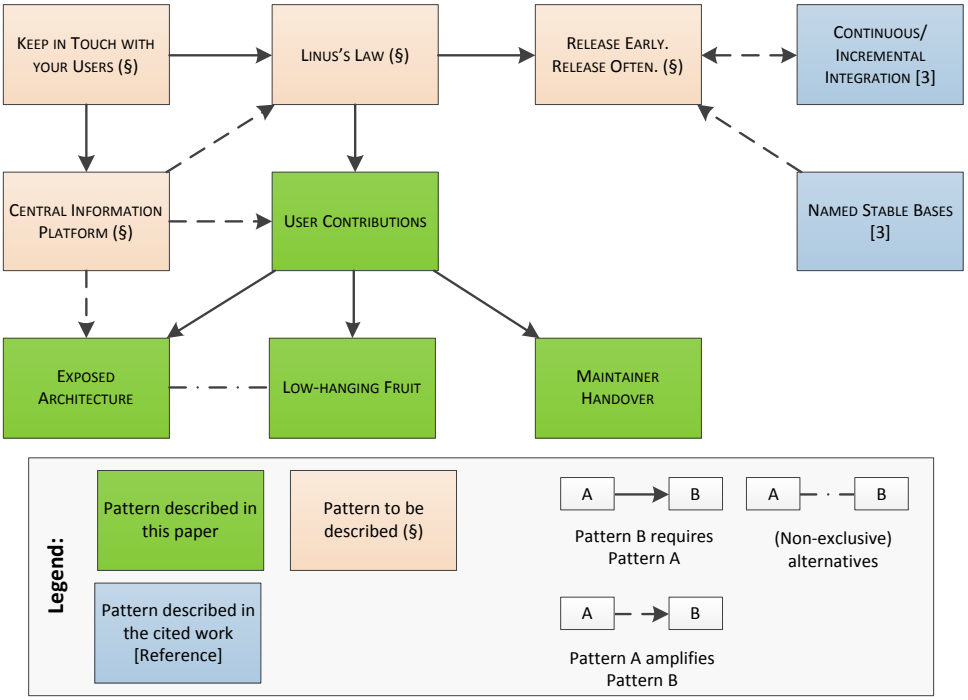

Figure 1: Pattern Map

2.1 User Contributions

Context

... when a project gathers momentum, a constant flow of user requests, questions and comments comes in.

Problem

The growing number of users correlates with a growing number of expectations, which the existing contributors cannot fulfill.

Forces

- The more users there are, the more different feature requirements they have.
- Users have inherent knowledge of what the project should deliver, but the developers cannot access this knowledge directly [1, chapter 15]. Some of the users are willing and able to develop patches that would satisfy each other's requirements.
- Users have little or no knowledge about the coding style and architecture the core developers have agreed upon.
- The core developers may have their own ideas where the project should be going but these ideas might be different from the other users' requirements.

Solution

Treat your users as co-developers and have them satisfy the users' demand.

There are several ways in which users may contribute to the project. Encourage your users to contribute to the project and clearly point out how to contribute. For example, have your users report bugs and request new features. Provide the users with a communication platform where they can provide support for each other. Users who want to fix bugs and extend the application's functionality must be able to access the source code of your software and they must be given the right to change the code. Describe how to build the software from the source code and provide tools that help with this task, like a download link to a preconfigured build environment. Explain your own requirements for code patches like coding style and design considerations.

When you receive contributions, take them seriously. If a patch has insufficient quality, help the contributor to improve the patch. "Stroke" your users whenever they send in patches and feedback[17].

Consequences

✓ When the number of users is large enough, than only a small fraction of users needs to contribute back in order to handle a significant share of the project's workload. Their contributions can easily equal or exceed the workload that the core developers contribute.

✓ Having users and developers in a double role allows projects to exist without "precise specifications or requirements documents" [8]. The requirements specification phase of the release cycle can be skipped, which enables more frequent releases and a higher fraction of the effort put into the project can be used for the actual development.

✗ The core developers now need to split time between reacting to input from users and pursuing their own scheme. They need to help adapting code contributions to the project's coding style and filter out the code that does not work together with the project's architecture.

✗ Users acquire influence on the software development and therefore the core developers lose some of their influence. The core developers need to open up for other people's ideas.

Examples

This pattern is one of the key elements of what Raymond described as the bazaar development style. Therefore, Raymond's own project "fetchmail" employed the pattern and also the Linux kernel, which was Raymond's primary object of observation. Other examples include the Mozilla projects.

Related Patterns

Linus's Law(§) is a special case of this Pattern and focuses on bug fixing. If you want a higher fraction of users to become developers, you should save some Low-hanging Fruit for beginners. New developers will find it easier to extend the software if you have an Exposed Architecture.

Sources

Gacek and Arief identified a characteristic "Developers are always users" and found it to be common among all FLOSS projects [8]. Other research suggests that in fact only part of the successful FLOSS projects employ this pattern: On the one hand, Krishnamurthy has shown that the number of developers in successful FLOSS projects increases over time – "The correlation between the age (in months) and the number of developers was 0.228". On the other hand, Figure 4 in his article reveals that the major part of FLOSS projects that have been running for longer than 20 months still do not have more than five developers [12]. This suggests that these projects have recruited none or only few developers since their foundation and thus, they have not successfully used the pattern User Contributions, at least not to the extent that users became developers. Raymond's advice number six is also a representation of this pattern: "Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging." [17].

Verification

First of all, a project subscribing to that pattern needs to provide some platform to interact with the users. It will be analyzed how contribution of unique users to mailing-lists, code-base, bug- and feature-tracker changes over time. Additionally, meta analysis of existing published statistics will be conducted.

2.2 Exposed Architecture

Context

As the project grows, software complexity increases and it becomes more and more difficult to understand the structure of the program.

Problem

Developers will not add features to the project when they do not know where to add their code.

Forces

- Active users who want to contribute to the project need some starting points but they have trouble finding the module where this functionality is best to be added. If they are not willing to spend a lot of their time to understand the software architecture, they either need help from the core developers or they give up before getting productive. Even if they do not give up, they have to spend time searching the code for the right place to add the new functionality.

- Software architecture is a well respected aspect in traditional software development. Development of the architecture is considered in the budget and there are architects dedicated to the task of delivering an architecture. An explicit architecture is necessary to plan further budgeting and the organization of the development teams. In a FLOSS project, the core developers will probably also have some idea about the architecture in mind but that idea is rarely explicitly documented since no formal budgeting is necessary and developers do not have to be strictly assigned to a team within the organization.

- Project quality decreases over time, as extensions and fixes will be inserted in suboptimal places.

- Following Conway's Law, the organizational structure of the project team is a representation of the software architecture [2]. Small project teams reorganize themselves and their software's architecture quickly when the need arises.

Solution

Publish the program's architecture.

You can publish the architecture in different places (see Figure 2): In a dedicated document, in a well documented source code or in the project's structure. Choose one of those places and publish this decision. Of course, combinations are also possible.

Dedicated documents for the software's architecture is probably the best way for outsiders to understand the structure and dependencies of the project. However, ensure that these documents are thoroughly maintained, otherwise they may damage more than they help.

Up to a certain project size, a self-describing source code is the least effort way to provide outsiders with an understanding of the program architecture. For larger projects, it is still a good idea to have self-describing source codes, so readers understand the structure inside each module. There are generally two complementary ways to create self-describing source codes. Firstly, literate programming advocates source code that is easy to understand without comments or other explanation. One aspect of this are variable and function names that accurately explain the function of this variable and function, respectively. Secondly, tools like JavaDoc[1]

[1] `http://www.oracle.com/technetwork/java/javase/`

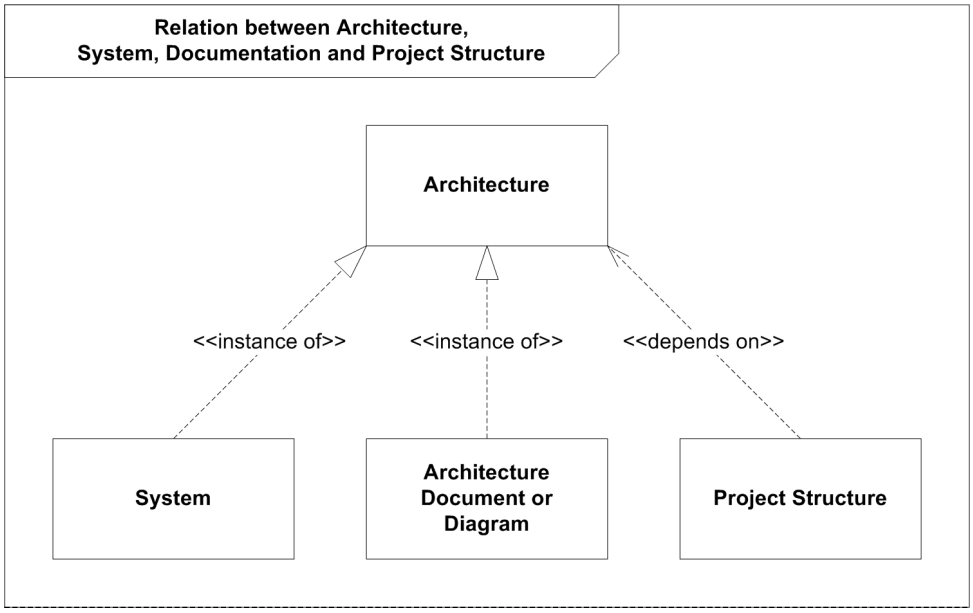

Figure 2: Relation between architecture, system, documentation and project structure

and Microsoft SandCastle[2] allow the creation of code documentation from source code and its comments. This requires extensive use of sensible comments, as there is no use to method-descriptions that do not say more than the method's and parameter's names already did.

Lay your project structure open. Show who is responsible for which part of the program. Provide your project sections with tools to publish information about themselves, allowing aspiring developers to overview the open tasks and contact the maintainer of the project section.

Consequences

- ✓ Prospective developers will find the place where to change code more easily. The number of developers increases, as the hurdle to join the project is lower. The project structure is self-organizing and therefore new developers are recruited more easily. These new developers use less time searching the code and discussing with the core develpers about the right place where to add patches. This saves time and less patches have to be dismissed when they do not fit into the software architecture.

- ✓ Developers are aware of the architecture and can potentially improve the architecture.

- ~ Understanding the architecture is key to find the best place to add new code. That prospective developers understand the architecture cannot guarantee that they find a place to add the demanded functionality though. The architecture can miss the interfaces necessary to add the new parts and changes to large parts of the program may be necessary. The developers may dismiss their contributions although they have understood the project and software architecture. Publishing the architecture does not help with this problem.

- ✗ Time spend on architecture cannot be used to develop new features and a formal structure might also suppress creativity.

Examples

The Unified Modeling Language (UML) modeling tool ArgoUML[3] documents its own architecture as UML diagrams[4]. Also, they document their process like issue handling[5] in UML diagrams.

Related patterns

The pattern User Contributions aims at promoting users to developers who therefore become part of the project's organizational structure. These new developers can integrate into the existing structure only if information about the structure is available.

It is not only sufficient to have an Exposed Architecure, it must also be clear to all prospective developers that it exists. Therefore, it is a good idea to provide a Central Information Platform(§) and promote the Exposed Architecture there.

Sources

Gacek and Arief have found out that some FLOSS projects have exposed architectures while others have closed architectures [8]. They call this property of a FLOSS project "Visibility of software architecture". An architecture can be kept secret intentionally, which may be another pattern – in case there are some forces that want to discourage others from joining the project. Or it can be unintentionally be closed if no developer ever formally wrote down the architectural considerations of the project.

The architecture of a software project has several different representations. When the project starts, the first developers have a mental representation of the architecture. This mental representation may be scripted in some formal way like UML. The source code surely represents the architecture as soon as it is written, although it may be more difficult to read the architecture just from the source code. Following Conway's Law, the organizational structure of the project team is another representation of the architecture [2].

Verification

As described in the pattern, an Exposed Architecture can be visible in explicit documentation, in well documented source code or in the organizational structure [2]. To fulfill the pattern, the project also needs to advertise that structure in some way. If a structure just seems to be present but is not mentioned at the project's site, the pattern is not met.

---

documentation/index-jsp-135444.html
[2] http://sandcastle.codeplex.com/
[3] http://argouml.tigris.org/
[4] http://argouml.tigris.org/wiki/Design
[5] http://argouml.tigris.org/wiki/The_big_picture_for_Issues

2.3 Low-hanging Fruit

Context

The project grows and gets more and more complex. The core developers grew up with the project's code and know how to handle it. Although you encourage User Contributions, the users cannot overview the project's structure and therefore they cannot work on the project.

Problem

No new developers join the project because there is too much to learn before they could be helpful.

Forces

- Software developers work on an application because they get paid for it or because they need the application for themselves. An additional reason may be the joy of programming in general but usually one of the other two reasons also applies. However, users contribute to the project if they are not only motivated but also capable to do so.

- If no additional money is to be invested and still new developers are wanted, the project must attract new developers in the group of users and potential users. The benefit for a FLOSS project is higher if the project recruits developers out of its user base that are already experienced developers. These developers may nevertheless stay users if they find all open issues in the project bug tracker, including their own issues, to difficult for them to fix. Even for experienced developers, this may be the case if their understanding of this specific project is still not very high.

- Most of the time, a project has some unfinished tasks. Easy tasks can be finished quickly and hard tasks tend to stay longer in the bug tracker.

- Developers spend only a little time on easy tasks and much time on hard tasks.

- When developers work on the open tasks of a project, easy tasks give a more immediate positive feedback and lead to quicker improvements. These tasks may therefore seem to be the ones to be worked on first. However, if the experienced developers quickly solve all easy problems, only the difficult ones are left for developers who want to join the project.

Solution

"Save some low-hanging fruit for beginners"[15].

Experienced developers of the project shall focus on the hard tasks that only they can solve. Leave some of the easy tasks open. Users are more motivated to work on these problems and therefore the transition from user to developer is more smoothly. Choose the tasks to be left open with care: They should not be so critical that the software quality suffers and they should not be so urgent that the core developers are forced to fix them because none of the users wanted to work on them in time.

You can encourage new developers to write and submit code for the software by announcing open easy problems on the project's web site. Name an experienced developer as contact who will provide help for developers willing to work on the announced problem.

If users contribute source code, either to existing bugs or for completely new features, encourage further contributions. If the source code has insufficient quality, core developers should not take over the bug and write new, higher quality source code themselves. Instead, they should point out the weaknesses of the original contribution, give examples of source code meeting the quality standards, and provide help with the tools that the other developers are using for this kind of problem. It is the primary goal to teach the user the abilities of a developer and fixing the bug is only secondary.

Consequences

✓ Users may be able to adapt the software for their needs and contribute their changes back to the project even if they have little development experience in the problem domain of the project. This is possible because the tasks are easy enough and they are provided tutoring.

✓ Over time, the users learn more and more about the project and contribute more regularly. The number of developers and their knowledge about the project increases.

✗ Some tasks seem to be easy, but they are difficult. These tasks will be unfinished for a long time. The software will have less features in the short term, since there are features that are easy to implement, which are left open on purpose.

∼ Core developers spend more time on difficult tasks, which is a better use of their valuable time. They might like the bigger challenges but they might also get discouraged because they only seldomly finish tasks.

✓ In the short term, core developers spend more time instructing prospective developers than they save by not working on the easy tasks. In the long term, there will be a higher number of developers with project knowledge and the project improves rapidly.

Examples

The Google Summer of Code[6] is a program that encourages students to work on FLOSS projects. Every participating project clearly defines a work package that can be solved in a three months time frame. The students are supported by mentors from the FLOSS projects that help to get started with the work. The students gain programming practice and the FLOSS projects get additional contributions. Also, the students may gain knowledge of the FLOSS project and stay valuable contributors of the project.

[6] `https://code.google.com/soc/`

Related Patterns

You should first rely on User Contributions and only if it is not possible for the users to contribute to your project, you should save some Low-hanging Fruit for beginners. Alternatively or additionally, you might document an Exposed Architecture, so your users can figure out how to fix the difficult tasks on their own, too.

Sources

One research thread in End-User-Programming tries to reduce the big leap from a user to a developer into a "gentle slope" (see [14], page 382). Naramore suggested to "Save some low-hanging fruit for beginners"[15], so new developers of FLOSS projects incrementally learn what is needed to participate in the project.

Raymond noticed in his first rule in "'The cathedral and the bazaar"' that software developers work better if they personally profit from their software: "1. Every good work of software starts by scratching a developer's personal itch." [17]. This means that there must still be itching problems before an unpaid developer starts working.

Verification

Verification of Low-hanging Fruit can be quite challenging, as a general accepted definition of easy tasks (i.e. "low hanging fruits") might not be possible. It is probably save to say that the pattern is met if a constant admission of new developers can be observed. A constant admission of new developers is not necessarily the same as a growing developer base, as others might leave the project at the same time. Then again, if no new admissions are observed – which is likely for many projects, given Krishnamurthy's [12] findings – a decision whether the pattern is met by a given project can only be done after careful examination of source code and the discussions in bug/feature trackers or mailing-lists.

2.4 Maintainer Handover

Context

For some reason, the leading maintainer / project owner lost interest in his project. Others might still be actively working on it, but it takes more and more time for him to respond to requests, some remain unanswered. Decisions are no longer made in a timely manner.

Problem

How can the development pace be kept when the project is about to lose its maintainer?

Forces

- On the one hand, comments from project maintainers that they are no longer maintaining the project are found in Sourceforge rarely. One the other hand, it is easy to find examples of abandoned projects. Some have reached a certain degree of maturity and attracted a community of at least some active users or even developers. The community is still interested in further development, but somehow the project owner is no longer as enthusiastic as he was once.

- Nevertheless the owner would rather see the project flourish independently than to see it die.

- There might be aspirants to take over the project. In a FLOSS project, any interested developer could just take over such a project by launching an own branch. However that might split the community and leave the users puzzling which branch to use, where to report bugs and feature requests to, etc. The replication of the project's infrastructure might not be trivial.

Solution

"When you lose interest in a program, your last duty to it is to hand it off to a competent successor." [17]

In a typical small project without an established base of active/core developers, finding a successor and making sure she brings the expertise needed will be the difficult part of the handover. If the leaving owner has not yet built a relationship of trust to some of the project's developers, then he might want to invest some additional time reviewing that developer's contributions before asking her if she was willing to take over the responsibility. However, if at this point of time there are no active developers contributing, then the owner might want to advertise the upcoming vacancy via the project's site and mailing lists. In that case, he is well advised to be patient – if no one feels a serious itch with the current version of the project, it can take a long time till a capable aspirant shows up. Then again, if no one feels a serious itch, then the project should not need much attention from the old owner, either.

Consequences

✓ A proper handover comes at low costs in time but smoothes the process of transition. The project and its community will benefit from a successful handover, since the development will continue; the newly won freedom and responsibility of the succeeding owner might bring fresh impulses and innovation. It also relieves the former owner of any burdens the project might otherwise impose on him.

✗ The only truly unfortunate outcome from the leaving owner's point of view can be time wasted on searching and building up a dedicated successor and be let down afterwards.

~ The old owner needs to be honest to himself about his declining interest and needs to be prepared to let go. There is also the risk of the new owner being not as competent or dedicated as needed. The project might suffer from a shift in direction and loss of vision. Such progress is not necessarily worse than an abandoned project, as it is still possible for discontent users to start a clean branch.

Examples

Robles and Gonzalez-Barahona have researched core maintainer turnover in large FLOSS projects [18] and found that the core group changed completely in some of the projects. They did not look at small projects with a single owner. There are examples for this pattern in use in smaller projects like TeXnicCenter[7] and CDex[8]. These examples also show varying difficulty of the handover.

Related patterns

The pattern User Contributions can help to attract developers who care about the project. These developers are potential candidates as the new maintainers.

Sources

Software developed by organizations can be categorized into three groups: Firstly, the software is either a proof of concept or prototype and is not intended to be used productively. Secondly, the software will only be used internally. Thirdly, the target audience is outside the organization. In the latter two cases, the organization has an interest to continue the development if the software is successful. Hence, organizations staff their successful software projects if this is profitable. This is not the case for FLOSS projects: Developers who lose interest in a FLOSS project will eventually quit the project even if the project is successful and there is demand on the user's side for the project. The pattern Maintainer Handover is therefore particularly important for FLOSS projects where user demand does not necessarily create the incentive to continue a project.

Raymond first described this behaviour as advice number five in his essay "The cathedral and the bazaar"[17].

[7] `http://www.texniccenter.org`
[8] `http://cdexos.sourceforge.net`

Verification

For each project, it will be analyzed whether there was a change of ownership of the project. Possible outcomes are

no change The pattern is not applicable.

smooth transition The pattern was successfully applied.

abandoned The pattern would have been applicable, but was not applied.

branched The pattern might have been useful, but instead of a handover, a new development branch was initiated.

complicated transition The project has been taken over, however the transition was not smooth but ended in or lead to conflicts. A significant share of users or developers seem to have left the project because of that. So the pattern was not successfully applied either.

3. CONCLUSION AND FURTHER WORK

This paper raises the assumption, that the so called FLOSS development process, which has been anticipated by numerous sources, can be interpreted in terms of patterns and pattern languages. Four patterns have been derived from related works and discussions about FLOSS in the communities. For each pattern exists a short paragraph describing how the pattern was identified and another paragraph about the possible verification of each pattern.

The next step will be the verification of the patterns. We will combine manual verification with an automated approach similar to the one described in [9]. During the identification of the patterns, some relations between the patterns already emerged. We expect that it will be possible to consolidate the patterns in a language or a family of languages during and after the verification. The FLOSS pattern languages can help to give a better insight into specific FLOSS projects and support comparisons of projects. Furthermore both FLOSS and proprietary projects might use these pattern languages to improve their process.

4. ACKNOWLEDGEMENTS

We thank Allan Kelly for the very valueable hints and ideas while being the shepherd for this paper. We also thank our colleague Bettina Biel for her motivating feedback to the early versions of the paper and we thank our Writer's Workshop group at PLoP for their feedback.

5. REFERENCES

[1] F. P. Brooks, Jr. *The Design of Design.* Addison-Wesley, 2010.

[2] M. E. Conway. How do committees invent? *Datamation*, 14(5):28–31, April 1968.

[3] J. O. Coplien and N. B. Harrison. *Organizational patterns of agile software development.* Pearson Prentice Hall, 2005.

[4] K. Crowston, H. Annabi, and J. Howison. Defining open source software project success. In *in Proceedings of the 24th International Conference on Information Systems (ICIS 2003)*, pages 327–340, 2003.

[5] A. Deshpande and D. Riehle. Continuous integration in open source software development. In B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, editors, *Open Source Development, Communities and Quality*, volume 275 of *IFIP International Federation for Information Processing*, pages 273–280. Springer Boston, 2008.

[6] K. Fogel. *Producing Open Source Software. How to Run a Successful Free Software Project.* O'Reilly Media, 2007.

[7] Free Software Foundation. The free software definition, 2008. http://www.gnu.org/philosophy/free-sw.html, last checked: 2010-05-24.

[8] C. Gacek and B. Arief. The many meanings of open source. *IEEE Software*, 21:34–40, 2004.

[9] D. German and A. Mockus. Automating the measurement of open source projects. In J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, editors, *3rd Workshop on Open Source Software Engineering / ICSE'03*, pages 63–68, Washington, DC, USA, 2003. IEEE Computer Society.

[10] K. Healy and A. Schussman. The ecology of open-source software development. Technical report, Department of Sociology. University of Arizona, 2003.

[11] A. Kelly. Patterns for technology companies. In *Proceedings of the 11th European Conference on Pattern Languages of Programs*, 2006.

[12] S. Krishnamurthy. Cave or community?: An empirical examination of 100 mature open source projects. *Social Science Research Network Working Paper Series*, 7(6), June 2002.

[13] B. Mishra, A. Prasad, and S. Raghunathan. Quality and profits under open source versus closed source. In *ICIS 2002 Proceedings*, pages 349–363, 2002.

[14] B. A. Myers, D. Smith, and B. Horn. *Languages for Developing User Interfaces*, chapter Report of the 'End-User Programming' Working Group, pages 343–366. A K Peters, 1992.

[15] E. Naramore. Why people don't contribute to os projects, and what we can do about it., March 2010. http://naramore.net/blog/why-people-don-t-contribute-to-os-projects-and-what-we-can-do-about-it, last checked: 2010-05-19.

[16] Open Source Initiative. The open source definition, 2004. http://www.opensource.org/docs/osd, last checked: 2010-05-22.

[17] E. S. Raymond. The cathedral and the bazaar. 2000. Revised version. http://www.catb.org/ esr/writings/cathedral-bazaar/cathedral-bazaar/, last checked 2010-05-24.

[18] G. Robles and J. Gonzalez-Barahona. Contributor turnover in libre software projects. In *OSS2006: Open Source Systems (IFIP 2.13)*, pages 273 – 286. Springer, 2006.

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

[20] J. Warsta and P. Abrahamsson. Is open source software development essentially an agile method? In J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, editors, *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 143–147. International Conference on Software Engineering, 2003.