Patterns for tearing down contribution barriers to FLOSS projects

Vincent Wolff-Marting, Christoph Hannebauer, and Volker Gruhn
paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, Gerlingstrasse 16, 45127 Essen, Germany
Email: {vincent.wolff-marting | christoph.hannebauer | volker.gruhn}@paluno.de

***Abstract*—The success of a "Free, Libre and Open Source Software" (FLOSS) project depends on its ability to attract new developers. However, before prospective developers can contribute their first patch, they have to overcome the contribution barriers of the FLOSS project. This paper presents two patterns. Each pattern identifies a possible problem in the contribution processes of FLOSS projects and shows a practice to alleviate this problem. The first pattern helps prospective developers compile the FLOSS project's source code and build an executable application. The second pattern encourages prospective developers to submit their modifications back to the FLOSS project and at the same time fosters integration of these modifications into the main development branch of the FLOSS project.**

I. Introduction

A common model of the community of a "Free, Libre and Open Source Software" (FLOSS) project is the onion model. Like the layers of an onion, the community of a FLOSS project also comprises layers. The maintainers and core developers of the FLOSS project, who steer the direction of the project, constitute the core layer. The next layer contains the co-developers who contribute source code. Active users, who report bugs and provide support to other users, represent the third layer. The fourth and outermost layer are the passive users who use the application of the FLOSS project but do not contribute anything back. The three innermost layers form the contributors of the FLOSS project. The closer contributors get to the core, the more important and profitable they are for the FLOSS project [1].

A FLOSS project grows if it succeeds to attract others for its cause. In the onion model, growth corresponds to a movement of people towards the core. People move to the core of a FLOSS project's community for various reasons. These reasons, the motivation of FLOSS developers, have been researched extensively [2], [3], [4]. There are still new research approaches to the motivation of FLOSS developers, though [5].

While the motivation pulls people towards the core of FLOSS projects, there are also opposed forces that keep them at distance from the core, as obviously there are not only core developers in FLOSS projects. These opposed forces are called contribution barriers [6]. Contributors must overcome these contribution barriers before they can move a layer towards the core of a FLOSS project. While much research on the motivation of FLOSS developers exists, there is still a research demand on contribution barriers.

There are different classes of contribution barriers. For example, there are psychological or sociological contribution barriers that depend on the openness of the FLOSS community. There are technical contribution barriers if the development environment or programming language are difficult for prospective developers to master. There may also be organizational contribution barriers if a FLOSS project has difficulties integrating new developers into its structure. There can even be juridical contribution barriers if the license terms of a FLOSS project do not meet the expectations of prospective contributors.

This paper describes two common contribution barriers encountered in FLOSS projects and possible methods to lower them. Sometimes, a contribution barrier cannot be lowered, but instead, the FLOSS project may shift the contribution barrier to the inside of the community's onion model. This is an improvement already, as contributors can move closer towards the core and therefore contribute more to the project.

The paper uses the pattern format that Alexander et al. introduced for the architecture domain [7]. The pattern idea has been adapted to several subdomains of software engineering [8], [9]. A pattern treats the problem and solution of a design problem in an abstract yet applicable way. Ideally, each pattern identifies a common invariant among a larger set of specific solutions to a problem. Related patterns form a pattern language. The patterns in a pattern language relate to each other, so some patterns must be applied before or after others.

Section II presents the current state of research on contribution barriers and patterns for FLOSS projects. Section III contains two patterns that maintainers of a FLOSS project use to lower the contribution barrier to their FLOSS project. Finally, Section IV summarizes the findings and shows opportunities for future research.

II. Related Work

A survey on 18 developers at International Business Machines Corporation (IBM) showed what orientation aids developers use when they join a software development project. Additionally, the survey showed obstacles that hampered the joining process. These obstacles correspond to the contribution barrier of FLOSS projects. The survey identified which obstacles exist and how they hampered the joining process. The obstacles include difficulties with the installation of the Integrated Development Environment (IDE), acclimatization to software tools specific for the given software development project, and insufficient documentation for software developers [10].

A case study on the FLOSS project Freenet first introduced the term "contribution barrier" as technical hindrances for

978-1-4799-0421-1/13/$31.00 ©2013 IEEE

newcomers. The survey identified four specific items as part of the contribution barrier, namely the difficulty to modify the source code, lacking familiarity with the programming language, a software architecture closed to new features, and highly coupled modules. This paper uses a broader definition of contribution barrier, including not only technical obstacles [6].

The parts of the contribution barrier that have been qualitatively identified are still subject to quantitative analysis [10]. One part of the contribution barrier that has been researched empirically is the choice of license. Early research vaguely indicated that FLOSS projects attract less contributors if they use restrictive licenses like the GNU General Public License (GPL) [11]. Afterwards, Comino et al. confirmed with a regression analysis of FLOSS projects on the software forge service Sourceforge.net[1] that restrictive licenses have a negative impact on the probability that a FLOSS project reaches a stable state [12]. However, recent empirical research contradicts each other: On the one hand, Subramaniam et al. show that the data on Sourceforge.net contains no significant signs that license choice influences attractiveness to developers [13]. On the other hand, Colazo and Fang conclude from the data on Sourceforge.net with very similar methods as Subramaniam et al. that restrictive licenses have a positive impact on developer attractiveness [14].

Existing approaches to reduce the contribution barrier to FLOSS projects encompass the application of wiki principles to software development. Wikis encourage contributions to natural language text artifacts. In our previous work, we suggest to use a Wiki Development Environment (WikiDE) that combines a web-based development environment with a wiki [15], [16].

Research has already identified a variety of patterns in the FLOSS context. Weiss presents nine patterns that allow commercial organizations to run projects as FLOSS projects and leverage benefits from this approach [17], [18]. More recently, Link contributes three additional patterns to this collection [19].

Link has also identified eight patterns on legal and licensing aspects of FLOSS projects. Some of these patterns apply to commercial organizations that run a FLOSS project, while others apply to FLOSS projects in general [20], [21].

Other patterns help run successful FLOSS projects, including technical aspects of software development in the special case of FLOSS development. Examples for these types of patterns are FREQUENT RELEASES [22], BUILD ON THE SHOULDERS OF OTHERS [22], PARALLEL DEVELOPMENT [22], EXPOSED ARCHITECTURE [23], and CENTRAL INFORMATION PLATFORM [24]. Patterns like MAINTAINER HANDOVER [23] and CROWD OF BUG DETECTORS [24] handle managerial aspects of FLOSS projects.

The patterns CREDIBLE PROMISE [22], OPEN DIALOG [22], LOW-HANGING FRUIT [23], and BAZAAR ARCHITECTURE [24] present existing methods to reduce the contribution barrier. These patterns belong to the same pattern language as the two patterns presented in this paper.

[1] https://sourceforge.net/

III. PATTERNS

This section contains two patterns that allow maintainers of FLOSS projects to lower the contribution barrier to their projects. There are multiple formats to record a pattern. This paper uses a format similar to Kelly's pattern format [25]. Each pattern comprises

- the *name* of the pattern,
- an *icon* symbolizing the pattern,
- the *context*, in which the pattern may be applied,
- a short description of the *problem* that the pattern solves,
- *forces* that users of the pattern have to consider, usually making the problem more difficult to solve,
- the *solution* of the problem,
- good and bad *consequences* of the solution that are labeled with a check mark and cross, respectively,
- the *known uses and sources* serving as examples for the application of the pattern, and
- *related patterns* that may rely on the pattern at hand.

Patterns have a target audience, in this case maintainers of FLOSS projects. The target audience may implement the solution described in the pattern and therewith solve the problem described in the pattern. Thus, patterns directly address this target audience, especially in the solution part.

Of course, this does not ban researchers and others who want to understand a specific problem domain to read patterns. In fact, patterns are sometimes considered a quick way to dive into a new problem domain.

A. Preconfigured Build Environment

Context: You are in an OPEN DIALOG [22] with the users and take advantage of their abilities as a CROWD OF BUG DETECTORS [24]. Now you want to encourage USER CONTRIBUTIONS [23].

Problem: **Joining developers get stuck when setting up the build environment.**

Forces:

- **Disproportionate configuration effort.** Joining developers want to modify an application's source code, but they have to build the application first. According to Fitzpatrick, the first build even is the hardest step in joining a software development project [26, p. 70f]. Dagenais et al. have also shown that installation of the development environment is a major obstacle for newcomers in a software development project [10]. Thus, for small modifications, joining developers might need

more time to configure the build environment than for the modification itself. The combined time effort might not be worth the benefit of the modification and so the joining developers may decide to not join the FLOSS project eventually [15].

- **Reuse creates dependencies.** Reusing existing libraries is often more effective than re-developing the required functionality. Reusing FLOSS libraries from other projects is therefore common and desirable among FLOSS projects. FLOSS projects can BUILD ON THE SHOULDERS OF OTHERS [22], but with every reused library there is an additional dependency. Additional dependencies complicate the build process and its setup becomes more difficult. As the dependencies may have other dependencies, the transitive closure of all dependencies can be much larger than the number of direct dependencies.
- **Platform diversity.** In company environments, a central Information Technology (IT) department often manages the configuration of all computers. Consequentially, the number of different configurations of the developers' computers is small and build configurations working on one machine are likely to also work on others. In FLOSS projects, however, the developers have no common computer configuration. Hence, build processes often require adaption to run on a specific computer.
- **Changing build processes.** As a FLOSS project evolves, it starts to depend on additional libraries, support new platforms, and acquire new features. Each of those changes also changes the build process. The FLOSS project flourishes, but developers that once mastered the build process and got their computers to build the application, constantly have to keep pace with these configuration changes. This requires a constant effort on the developers' side. Developers that only have little spare time to put into the project are left behind.
- **Prior experience.** Joining developers may have experience with similiar software development projects. They profit from this experience and maybe they can even partly reuse their existing build systems to build the application of the joined FLOSS project. This decreases the effort to set up the build environment, but it does not change the effort for joining developers without experience in the specific technology used by the joined FLOSS project.
- **Interpreted languages.** Interpreters execute source code without building binaries. For example, most standard web browsers contain interpreters to run JavaScript programs. No build environment must be set up for FLOSS projects written in JavaScript, but many FLOSS projects are written in other languages, as some types of applications are difficult or impossible to write in interpreted languages. Examples are Operating System (OS)s that must be compiled to machine code and virtualization environments that themselves interpret JavaScript or similar languages.
- **Nobody in charge.** Joining developers suffer from build configurations that are difficult to set up, but they are not competent enough to simplify these build configurations. Core developers have the abilities to simplify the build configurations but they suffer only little from those build configurations, as they have configured their build environment already. Nobody has both the personal motivation and ability to simplify the build configurations and, thus, the build configurations stay difficult to set up.

Solution: **Provide a Virtual Machine with a preconfigured build environment.**

Put all dependencies and required build tools into one package, so interested developers have to download only a single file to build your application. You should strive to minimize configuration time for the build environment. Still, there are multiple alternatives to realize this requirement, depending on the type of application your FLOSS project develops.

As the first alternative, you may in fact set up a build environment on a Virtual Machine (VM) and ensure that this build environment builds your application without any additional configuration. You can then offer the VM as a download on the FLOSS project website. If possible, use a widely accepted format for the VM, so all interested developers may start the VM on their host machines.

The second alternative is only a subsystem instead of a whole VM. Examples for such subsystems are Cygwin[2] and MinGW/MSYS[3]. These subsystems provide Linux functionality on Microsoft Windows. Because such a subsystem is still difficult to configure, you can preconfigure a subsystem with the tools and dependencies required to build you FLOSS project's application and offer it as a download on the FLOSS project's website.

Third, an even leaner solution is a portable IDE as the download package. This is of course only possible if an appropriate portable IDE exists for the programming language and environment of the FLOSS project. Additionally, all required libraries and other dependencies must be packaged into one package with the IDE—again, this is not possible for all programming languages and environments. Such a portable IDE depends only on data within the package. For this solution, it may be necessary to package IDEs in multiple different configurations. For example if every IDE package supports only a single OS, but the developers of the FLOSS project use different OSs, you should provide one IDE package for every OS.

For some FLOSS projects, a well made Makefile [27], a Microsoft Visual Studio project file [28], or other type of build script suffices. This is arguably a fourth alternative to implement this pattern. Instead, these cases may be considered FLOSS projects that do not need to implement this pattern, as their native build processes are simple enough already. A build script is usually the first step to carry out one of the other alternatives.

[2]http://cygwin.com/
[3]http://www.mingw.org/wiki/MSYS

Consequences:

✓ **Lowered contribution barrier for co-developers.** Build VMs ease the build of the application. Joining developers experience lower barriers for their contribution and are therefore more likely to also contribute smaller modifications that have not been worth the effort before. The FLOSS project therefore attracts more co-developers.

✓ **One package for everything.** The build VM contains all dependencies of the FLOSS project along with the other build tools. Joining developers can immediately build the application without searching for and building dependencies first.

✓ **Cross-platform support.** Build VMs provide an additional layer between the build scripts and the developer's host OS. Therefore, the developers' host OSs are less important now and developers may run any OS and still join the FLOSS project.

✗ **Outdated build VMs.** Maintaining build VMs takes time. If the core developers neglect this maintenance, the build VMs become outdated and they cannot build the latest versions of the application [29].

✗ **VM proliferation [30].** Developers who are active in a high number of FLOSS projects or subprojects may have to store a lot of build VMs from each of these projects or subprojects. They may lose track of their build VMs and spend time searching for the right build VM for their current project. As a consequence of this confusion, they may download the same build VMs multiple times which even worsens the problem.

✓ **Easy access to complex applications.** Build VMs compile even complex applications without much configuration effort if they are preconfigured correctly. Joining developers have little trouble to modify these complex applications.

✓ **Contribution barrier for core developers.** While build VMs suffice for smaller source code modifications, they may lack the flexibility needed for larger source code modifications. Developers still need to get in touch with the details of the build processes if they want to become core developers. Build VMs are still an intermediate step for becoming a core developer and make the slope more gently, although they do not lower the overall work necessary.

Known Uses and Sources: For Mozilla Firefox developers using Microsoft Windows as their operating system, Mozilla provides a package called MozillaBuild. MozillaBuild contains development tools, the MSYS environment to provide Unix functionality under Windows, and the programs necessary to compile the Firefox source code, among other things [31].

Another example are the preconfigured download packages for the Eclipse IDE. At least for simple plugins, the package *Eclipse for RCP and RAP Developers* contains all prerequisites to work on the source code of plugins [32].

Related patterns: A successful build is only the first step to more USER CONTRIBUTIONS [23]. Joining developers may fail to fix complicated faults or add advanded features at the beginning. These joining developers need LOW-HANGING FRUIT [23] to get started with the development on the FLOSS project. If the FLOSS project has a BAZAAR ARCHITECTURE [24], joining developers will integrate their modifications into the existing source code more easily—and if the FLOSS project also has an EXPOSED ARCHITECTURE [23], they will even know where to integrate these modifications.

B. Unit Tests for Contributors

Context: Your FLOSS project supports users to modify the application's source code. A PRECONFIGURED BUILD ENVIRONMENT reduces technical contribution barriers, while a BAZAAR ARCHITECTURE [24] and possibly LOW HANGING FRUIT [23] reduces conceptual barriers for prospective developers. Your users therefore fix bugs and develop new features for the application.

Problem: **Joining developers hold back their patches as they are afraid to introduce new bugs.**

Forces:

- **Fear of failure.** Some of the application's users invest their time and energy to write patches for the application. They are willing to give their patches to the community, but they are afraid to publish erroneous patches, as it may be embarrassing for them if others can see their mistakes.
- **Agile architecture.** The developers are also users. If the developers find a new use case for the application, they might immediately start to develop an additional feature for the application, changing the architecture and interfaces on the way. This allows the FLOSS project to quickly react to emerging market demands, but the FLOSS project must be managed and the software architecture designed in a way that allows developers to change the application design quickly in any direction – even if the developers had only been users before they started to work on the new feature.
- **Platform specific bugs.** Developers may intensively test a patched application on their platforms, but this does not ensure that the patches are free of errors. The patch can be incompatible to platforms that the developers had no access to for their tests.
- **Test effort.** Developers may quickly run superficial, manual tests, but it requires much more time to write automated unit tests.
- **Unpredicted side effects.** After developers have modified the source code, they want to check whether their modification behaves as intended. For the bug fix of an easily reproducible bug, this check guarantees that the modification really fixes the bug as it was supposed to. But even in this simple case, the modification may still

cause unpredicted side effects that induce new bugs in the application.

- **Understanding patches.** If there are enough beta testers or co-developers, they will quickly find bugs that recent modifications have induced, but then the users who find the bug need to put effort in reproducing the bug and finding the cause of the issue. Even if they can track down the bug to a single patch, they still have to understand the rationale of the patch before they can fix the bug [33].

Solution: **Provide easy access to unit tests even for prospective developers.**

For applications that run on a variety of platforms and configurations, automated unit tests are the only way to have at least some basic checks that these platforms and configurations are still working. A single developer cannot cope with more than a few different platforms. In these cases, provide test servers for all supported platforms and configurations. Even if it is a high effort at the beginning, this pays off quickly, as manual tests may be reduced.

If there are only a few or maybe only one platform, unit tests are less important. However, the minimum implementation of this pattern requires automatic builds of the application in regular intervals, so at least compile time compatibility is ensured. This is best achieved with a Continuous Integration (CI) systems like Jenkins[4], Hudson[5], or CruiseControl[6]. These systems support different test frameworks, so later addition of automated tests is easy.

However, it is insufficient to restrict CI services to the core developers and the main development branch. The earlier a bug is detected, the easier it is to fix [34]. Therefore, co-developers shall also be able to run unit tests. Include the code for unit tests into your PRECONFIGURED BUILD ENVIRONMENT, so co-developers can run these unit tests locally on their machines. A PRECONFIGURED BUILD ENVIRONMENT is also a good starting point for the configuration of a CI system, since you can reuse your build scripts.

Consequences:

✓ **Confident developers.** As all patches are tested before they are published, their developers are confident that the patches do not contain obvious bugs. The patch developers know that even if the patches introduce new bugs, it's a joint mistake, as also the test developer has missed a test case. Joint mistakes are less embarrassing and submitting a patch is less fearsome.

✗ **Test dependencies.** Although the tests are not delivered as part of the application release and they are not needed for the actual functionality, they are still part of the software architecture. The tests therefore add complexity to the architecture and all main components have tests that depend on them. Thus, interface or architectural changes now also require changes to the tests. This reduces flexibility as more work is required for such types of changes. Also, the existing tests cannot detect flaws introduced with these types of changes, because new interfaces require new tests.

✓ **Platform compatibility.** Automated build systems create ports of the application for all target platforms. Thus, the applications are tested on all of these target platforms and the application is guaranteed to provide at least the tested functionality on all platforms.

✓ **Tests are examples.** Beside the increased stability, developers benefit from tests if they use the test source code as example source code for other purposes. The effort required to write the tests therefore pays off faster.

✓ **Second line of defense.** Patches may induce side effects not detected by any test. In this case, the tests have been useless. However, there are cases where the tests detect problems that without the tests would have gone into the release of the application. Overall stability is therefore increased.

✓ **Fix your own bug.** Some bugs in the patches are now detected early, so the original developers of the patches may fix these bugs before any other developers come into contact with the bug. As the original developers of the patches know the rationale and the structure of the patch, they need much less effort to fix the bugs than other developers would have needed.

Known Uses and Sources: Mozilla tests all changes in their code base automatically [35]. Travis-CI[7] and BuildHive[8] provide CI with tests for FLOSS projects hosted on github. Projects like The Legion of the Bouncy Castle[9] include unit tests in their source code packages. Facebook provides unit tests for developers of facebook apps and plugins, like tests for iOS apps[10] and for PHP: Hypertext Preprocessor (PHP) plugins[11].

Related patterns: Unit tests allows fast and automated tests of the application. This is a precondition for FREQUENT RELEASES [22], as every release requires tests. FREQUENT RELEASES would require much manual effort if no automated tests are in place. Consider more intensive test for the less frequent, official releases in a PARALLEL DEVELOPMENT [22] strand.

IV. CONCLUSION AND FUTURE WORK

This paper presented the two patterns PRECONFIGURED BUILD ENVIRONMENT and UNIT TESTS FOR CONTRIBUTORS. Both identify a problem and its solution in the context of managing FLOSS projects. The patterns help lower the contribution barrier for prospective developers. The former pattern reduces the effort to build the application of the FLOSS project and therefore encourages modifications to the source code of the FLOSS project. The latter pattern reduces the risk of submitting these modifications back to the FLOSS project.

[4] http://jenkins-ci.org/
[5] http://www.hudson-ci.org/
[6] http://cruisecontrol.sourceforge.net/
[7] https://travis-ci.org/
[8] https://buildhive.cloudbees.com/
[9] http://www.bouncycastle.org/
[10] https://github.com/facebook/facebook-ios-sdk/tree/master/src/tests
[11] https://github.com/facebook/facebook-php-sdk/tree/master/tests

As explained in Section II, future research should analyze what problems amount to the contribution barrier and how much these problems amount to the contribution barrier. As an example, current research yields ambiguous results on whether restrictive licenses are a contribution barrier or not.

In addition to the patterns presented in this paper and in our previous work [23], [24], the concept of a WikiDE also tries to lower the contribution barrier to FLOSS projects [15], [16]. In our ongoing research, we will continue to work on the WikiDE concept and try to identify additional patterns that help lower the contribution barrier.

REFERENCES

[1] K. Crowston, H. Annabi, J. Howison, and C. Masango, "Effective work practices for floss development: A model and propositions," in *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2005, p. 197.1.

[2] A. Hars and S. Ou, "Working for free? motivations of participating in open source projects," in *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, jan. 2001, p. 9 pp.

[3] K. Lakhani and R. G. Wolf, "Why hackers do what they do: Understanding motivation and effort in free/open source software projects," *Social Science Research Network*, pp. 1–27, 2003.

[4] S. Krishnamurthy, "On the intrinsic and extrinsic motivation of free/libre/open source (floss) developers," *Knowledge, Technology, & Policy*, vol. 18, pp. 17–39, 2006.

[5] H. Benbya and N. Belbaly, "Understanding developers' motives in open source projects: A multi-theoretical framework," *Communications of the AIS*, vol. 27, pp. pp. 589–610, January 2010.

[6] G. von Krogh, S. Spaeth, and K. R. Lakhani, "Community, joining, and specialization in open source software innovation: A case study," *Research Policy*, vol. 32, no. 7, pp. 1217–1241(25), July 2003.

[7] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*. New York: Oxford University Press, 1977.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

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

[10] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. P. de Vries, "Moving into a new software project landscape," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 275–284.

[11] K. Stewart, A. P. Ammeter, and L. M. Maruping, "Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects," *Information Systems Research*, vol. 17, no. 2, pp. 126–144, 2006.

[12] S. Comino, M. Manenti, Fabio, and L. Parisi, M., "From planning to mature: On the success of open source projects," *Research Policy*, vol. 36, pp. 1575–1586, 2007.

[13] C. Subramaniam, R. Sen, and M. L. Nelson, "Determinants of open source software project success: A longitudinal study," *Decision Support Systems*, vol. 46, no. 2, pp. 576 – 585, 2009.

[14] J. Colazo and Y. Fang, "Impact of license choice on open source software development activity," *Journal of the American Society for Information Science and Technology*, vol. 60, no. 5, pp. 997–1011, 2009.

[15] V. Gruhn and C. Hannebauer, "Using Wikis as Software Development Environments," in *Proceedings of the 11th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT_12)*, ser. Frontiers in Artificial Intelligence and Applications. IOS International Publisher, October 2012.

[16] ——, "Components of a wiki-based software development environment," in *E-Learning, E-Management and E-Services (IS3e), 2012 IEEE Symposium on*, 2012.

[17] M. Weiss, "Profiting from open source," in *Proceedings of the 15th European Conference on Pattern Languages of Programs*, ser. EuroPLoP '10. New York, NY, USA: ACM, 2010, pp. 5:1–5:8.

[18] ——, "Profiting even more from open source," in *Proceedings of the 16th European Conference on Pattern Languages of Programs*, ser. EuroPLoP '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:7.

[19] C. Link, "Patterns for the commercial use of Open Source: Economic aspects and case studies," in *EuroPLoP 2012*, no. Version 398, 2012.

[20] ——, "Patterns for the commercial use of open source: legal and licensing aspects," in *Proceedings of the 15th European Conference on Pattern Languages of Programs*, ser. EuroPLoP '10. New York, NY, USA: ACM, 2010, pp. 7:1–7:10.

[21] ——, "Patterns for the commercial use of Open Source: License patterns," in *EuroPLoP 2011*, no. Version 331, 2011.

[22] M. Weiss, "Performance of open source projects," in *Proceedings of the 14th European Conference on Pattern Languages of Programs*, 2009.

[23] C. Hannebauer, V. Wolff-Marting, and V. Gruhn, "Towards a Pattern Language for FLOSS Development," in *Proceedings of the 2010 Conference on Pattern Languages of Programs*, Hillside Group. New York: ACM, 2010.

[24] ——, "Contributor-Interaction Patterns in FLOSS Development," in *EuroPLoP 2011*, Jul. 2011.

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

[26] P. Seibel and B. Fitzpatrick, "Brad Fitzpatrick," in *Coders at Work*, P. Seibel, Ed. Berkely, CA, USA: Apress, 2009, ch. 2, pp. 49–90.

[27] Free Software Foundation, Inc., "GNU Make," Jul. 2010, [accessed 2013-03-28]. [Online]. Available: https://www.gnu.org/software/make/

[28] Microsoft Developer Network, "MSBuild Project File Schema Reference," Microsoft, 2013, [accessed 2013-03-28]. [Online]. Available: http://msdn.microsoft.com/en-us/library/5dy88c2e.aspx

[29] Mozilla Foundation. (2009, Jul.) Mozilla Bug 459257 - MozillaBuild fails on WinXP x64 due to errors in guess-msvc/start-* scripts. [accessed 2013-03-29]. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=459257

[30] M. Price, "The paradox of security in virtual environments," *Computer*, vol. 41, no. 11, pp. 22–28, 2008.

[31] Mozilla Developer Network. (2013, Feb.) Windows build prerequisites, Section MozillaBuild. Mozilla Foundation. [accessed 2013-03-25]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Developer_Guide/Build_Instructions/Windows_Prerequisites#MozillaBuild

[32] The Eclipse Foundation. (2013, Mar.) Eclipse Downloads. [accessed 2013-03-27]. [Online]. Available: http://www.eclipse.org/downloads/

[33] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How Do Software Engineers Understand Code Changes? An Exploratory Study in Industry," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012)*. Research Triangle Park, North Carolina, November 2012.

[34] M. Fowler. (2006, May) Continuous integration. [accessed 2013-03-28]. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html

[35] Mozilla Developer Network. (2013, Mar.) Mozilla automated testing. Mozilla Foundation. [accessed 2013-03-05]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Automated_testing